Q# as a Quantum Algorithmic Language

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

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

Sarah Marshall
Microsoft Quantum
sarah@sarahmarshall.name

Robert Rand
University of Chicago
rand@uchicago.edu

Q# is a standalone domain-specific 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 language definition for Q#, placing the language on a solid mathematical foundation and enabling further evolution of its design and type system. This paper presents $\lambda_{Q\#}$, an idealized version of Q# that illustrates how we may view Q# as a quantum ALGOL (algorithmic language). We show the safety properties enforced by $\lambda_{Q\#}$'s type system and present its equational semantics based on a fully complete algebraic theory by Staton.

1 Introduction

Microsoft's Q# programming language [57] is one of the most full-featured quantum programming languages that have emerged from the recent boom in quantum computing research. But with a growing code base and increasing popularity comes the demand for more features and the resulting added complexity. Hence, Q# faces challenges familiar to many growing programming languages—maintaining correctness, ease of use, and intuitive understanding while evolving to meet users' needs.

Quantum programming languages face unique challenges that are not present in classical languages. Quantum algorithms are more challenging to design and reason about than classical ones as they use quantum phenomena like superposition and entanglement. Quantum programs are difficult to test and debug. Their simulation on classical computers is slow and limited to a handful of qubits while languages like Q# are designed for large-scale fault-tolerant quantum computers with thousands of logical qubits. When running a quantum program directly on quantum hardware, we cannot observe the whole quantum state directly, and measuring a classical result during execution (partial observation) may itself destroy the state. Additionally, existing quantum hardware provides limited qubit count and poor gate fidelity.

These challenges underscore why it is essential for Q# to have a well-specified definition that can serve as a foundation for extensions, multiple implementations, and formal verification of programs written in the language. A formal specification and mechanization of its metatheory 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 core version of the language, provide an elaboration from the surface language to this core language, and provide static and dynamic semantics for the core. In this paper, 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. We call this core $\lambda_{Q\#}$. In it, we make several implicit features of Q# explicit: its treatment of qubits as references, its stack-like memory management that enables reasoning about the quantum state in a local manner, and its safe synthesis of effectful and pure computation. In the classical setting, this stack-like memory management and combination of effectful and pure computation are inherent to many ALGOL-like languages [48].

Submitted to:
QPL 2022

Contributions In 2018, when introducing Q#, its designers stated that as opposed to several existing *circuit definition* languages, "Q# is an *algorithm definition* language" [57]; the goal of our paper is to show that, in its essence, Q# is a quantum *algorithmic language* (ALGOL).

- To support this characterization, we introduce $\lambda_{Q\#}$, an idealized version of Q# inspired by Harper's language **MA** (Modernized Algol) [16]. In $\lambda_{Q\#}$, we expose values of the `Qubit` type in Q# as references to logical qubits and formalize the ALGOL-like stack discipline implicit in Q#'s quantum memory management.
- We develop a type system for $\lambda_{Q\#}$ that extends Q#'s type system to enforce the no-cloning theorem and stack-like management of qubits.
- We provide an equational dynamics for $\lambda_{Q\#}$ building upon the fully complete equational theory of quantum computation by Staton [55].
- Finally, we provide an elaboration relation from Q# to $\lambda_{Q\#}$, thereby endowing a significant portion of Q# with a formal specification and additional safety guarantees.

Outline In the rest of the paper, we review background on the Q# programming language and Staton's theory for quantum computation (§2); introduce $\lambda_{Q\#}$ along with its syntax and semantics (§3); describe how $\lambda_{Q\#}$ is faithful to the surface Q# language (§4); and discuss related and future work (§5 and §6).

2 Background

Before introducing $\lambda_{Q\#}$, we discuss the two projects that inspired our work. The first is Microsoft's Q# [57], a modern, self-contained quantum programming language that boasts a large community of developers. The second is Sam Staton's equational theory for quantum programs [55], which provides a compelling alternative to the standard matrix-based semantics for quantum programs.

2.1 The Q# Programming Language

Q# [57] 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 rather than circuits, where quantum operations can be combined with classical control flow such as branches and loops. When a programmer measures a qubit, they can perform an arbitrary classical computation on the result, and the program execution can continue without requiring the qubit to be released. This computational model 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. Operations can call functions, but functions cannot call operations. An example Q# program, implementing the quantum teleportation protocol, is shown in Listing 1 in Appendix D.

Q# contains a blend of functional and imperative features (it evolved from an F#-like language [4]). For classical data, Q# follows the so-called value semantics [21], perhaps better known as referential transparency. Q# functions are always pure, and variable bindings are immutable by default. Bindings may be declared `mutable`, but they correspond to a local state change, enclosed in the scope of the parent callable. Hence, equational reasoning is possible across function boundaries. By contrast, qubits are opaque types that act as references to logical qubits [10, 11]—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` after altering the quantum state.

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

(a) Returning a qubit after its lifetime has ended.

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

(b) Using the same qubit as both control and target.

Figure 1: Sample unsafe Q# programs.

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 parameterized by quantum subroutines are easily expressed in Q# using higher-order operations. For example, an operation implementing Grover's search [14] can accept an 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 and controlled versions of unitary operations. Operations can declare their support for `Adjoint` and `Controlled` *functors* (in Q#'s terminology[1]) using the *characteristics* `Adj` and `Ctl`, respectively.

Q# follows the QRAM model of computation [26], which assumes an unbounded supply of logical qubits from which the programmer can obtain a reference to a new qubit by calling the `use` command. Qubits are hence allocated and deallocated in a stack-like manner, where the lifetime of a qubit is equivalent to the lexical scope of the `use` command. Even though this stack discipline can ensure safe (quantum) memory management, it is currently not enforced by the Q# compiler and type system. Figure 1a shows a minimal example that passes the type checker but fails at runtime (in a simulator).

Programmers are allowed to create new bindings using `let` that refer to the same qubit as another binding, leading to *aliasing* of qubit references. While aliasing is ubiquitous in Q#, it can lead to unsafe behavior in violation of the *no-cloning theorem* [61], which forbids duplication of qubits. In Figure 1b, both `q1` and `q2` 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, Q# cannot prevent this issue statically.

An informal specification of the Q# language was recently published [22]. However, it does not capture the subtle aspects of the language, such as the aliasing of qubit references or its goal of maintaining a stack discipline. Our work makes these subtleties explicit and formal.

2.2 An Equational Theory for QRAM

Staton [55] presents a substructural (linear) version of his framework for "parameterized algebraic theories" [54]. He develops an axiomatization for quantum computation using this framework, which he shows to be fully complete. Staton then extracts an equational theory for a quantum programming language from his algebraic theory that uses *generic effects* rather than *algebraic operations* [44]. Finally, Staton remarks upon a variant of his theory [55, §6.2] that applies to the QRAM model, where instead of working with qubits, we work with references to qubits. This is the approach taken in projects like the Quantum IO Monad [2], Quantum Hoare Type Theory [51, 52], and, to our advantage, Q#.[2]

Here we reproduce Staton's theory of a "quantum local store" [55, §6.2, p. 11] for reference; we will

[1] Perhaps a better name for functors would be 'combinators' from the functional programming community to avoid confusion with other accepted meanings of the term 'functor.'

[2] However, the stack-like management of qubits is unique to Q#.

see in §3.3 how this algebraic theory helps us describe the equational dynamics of $\lambda_{Q\#}$. We assume that the qubit references are unique, which we guarantee for $\lambda_{Q\#}$ in §3.2.1.

Generic Effects Staton adds the following generic effects to a standard linear type theory and obtains a quantum programming language [55, §5] similar to Selinger's QPL [50].

$$\frac{}{\vdash \underline{\mathsf{new}}() : \mathrm{qubit}} \qquad \frac{\Gamma \vdash t : \mathrm{qubit}^{\otimes n}}{\Gamma \vdash \underline{\mathsf{apply}}_U(t) : \mathrm{qubit}^{\otimes n}} \qquad \frac{\Gamma \vdash t : \mathrm{qubit}}{\Gamma \vdash \underline{\mathsf{measure}}(t) : \mathrm{bool}}$$

Program Equations There are two interesting classes of axioms (ignoring the axioms that describe commutativity of let). For completeness, we also show axiom (C) pertaining to the *discard* operation (equivalent to measuring a qubit and ignoring its result), but it does not apply in the QRAM model as noted by Staton [55, §6.2].
Axioms relating unitary gates and measurement:

$$(A) \qquad \underline{\mathsf{measure}}(\underline{\mathsf{apply}}_X(a)) \equiv \neg\, \underline{\mathsf{measure}}(a)$$

$$(B) \qquad \text{let } (a',x') \text{ be } \underline{\mathsf{apply}}_{D(U,V)}(a,x) \text{ in } (\underline{\mathsf{measure}}(a'),x') \equiv$$
$$\text{if } \underline{\mathsf{measure}}(a) = 0 \text{ then } (0,\underline{\mathsf{apply}}_U(x)) \text{ else } (1,\underline{\mathsf{apply}}_V(x))$$

$$(C) \qquad \underline{\mathsf{discard}}(\underline{\mathsf{apply}}_U(x)) \equiv \underline{\mathsf{discard}}(x)$$

Axioms relating allocation with unitaries and measurement:

$$(D) \qquad \underline{\mathsf{measure}}(\underline{\mathsf{new}}()) \equiv 0$$

$$(E) \qquad \underline{\mathsf{apply}}_{D(U,V)}(\underline{\mathsf{new}}(),x) \equiv (\underline{\mathsf{new}}(),\underline{\mathsf{apply}}_U(x))$$

where $D(U,V) = U \oplus V = \left(\begin{smallmatrix} U & 0 \\ 0 & V \end{smallmatrix}\right)$ applies U or V depending on the value of its first argument.

Axiom (A) says that applying the quantum X gate to a qubit and then measuring it is the same as negating the measurement result. Axiom (B) explains the action of a block diagonal matrix $D(U,V)$ as quantum control by stating that applying the diagonal matrix and then measuring the control qubit is equivalent to measuring the control qubit and branching on the result to decide whether to apply U or V. Axiom (C) says that if the qubits are to be discarded, applying a unitary is the same as doing nothing. Axiom (D) states that measuring a new qubit always results in 0, i.e., qubits are always initialized to 0. Axiom (E) says that using a new qubit as control is the same as controlling by 0.

We will show in §3.3 that our $\lambda_{Q\#}$ calculus follows similar program equations.

3 $\lambda_{Q\#}$: A Core Calculus for Q#

Our approach closely follows the type-theoretic interpretation of Standard ML, where Harper and Stone [19] developed a well-typed internal language for Standard ML, defined an elaboration relation between the external language and this internal language, and proved the properties of the metatheory of the language using the internal language. Harper and collaborators [7, 28] followed this work with the mechanization of the metatheory using the Twelf logical framework [43]. 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, we define an elaboration relation from the surface Q# language to $\lambda_{Q\#}$ (§4). 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 tractable because we only need to do it for the small, well-formed core.

To mirror the separation between operations and functions in Q#, we base the design of $\lambda_{Q\#}$ on Harper's **MA** (Modernized Algol) [16], which maintains a separation between commands that modify state and expressions that do not. Q# is an ALGOL-like language in more ways than one—syntax, block-structure, local (classical) state, and safe integration of functional and imperative paradigms. However, unlike Reynolds' Idealized ALGOL [48], the variables in Q# are immutable by default, and the language follows a call-by-value semantics, both of which make it closer to Harper's **MA**.

Before presenting $\lambda_{Q\#}$, let us motivate our design choices and establish some terminology. Q# has two kinds of variable bindings. 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. Significantly, they are restricted to the lexical scope in which they are bound. Since they do not affect equational reasoning across function boundaries, and our focus is on the quantum state, we ignore `mutable` variables in the rest of this paper. Qubits have type `Qubit` and syntactically look just like other variables but are references to underlying logical qubits that are never exposed. Unlike classical bindings, which follow value semantics, aliasing is permitted on qubits, leading to problems such as the violation of the no-cloning theorem discussed in §2.1. Qubits come into scope with either the `use` or `borrow` keywords. The former provides access to freshly allocated qubits in state $|0\rangle$, while the latter allows access to previously allocated (and potentially entangled) qubits. We do not consider borrowing in this work as it is an optimization concern that lets a programmer reuse ancillae in their code. The only allowed operations on qubits are gate application and measurement.

3.1 Syntax

Figure 2 presents the abstract syntax of $\lambda_{Q\#}$. We divide the grammar into a monadic effectful command language and a pure expression language (the simply-typed λ-calculus extended with encapsulated commands). We precisely specify the binding structure of the syntax following the notion of abstract binding trees from Harper's **PFPL** [17]. Following the **PFPL**-syntactic conventions, $\texttt{qref}\langle q\rangle$, $\texttt{gateap}\langle U_{2^n}\rangle(e)$, and $\texttt{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1\,;e_2)$ are indexed by symbols [17, Ch. 31] (marked in color) and variadic product operators are indexed by finite sets, n, where we slightly abuse the notation, $n \triangleq \{1,2,\ldots,n\}$. We also use the notation, $\tau_n \triangleq i \hookrightarrow \tau_i \mid i \in n$. Some operators take optional arguments marked by square brackets. We will often use the concrete syntax in blue color, and some standard derived forms from Harper's language **MA**, shown in Appendix A, wherever there is no possibility of confusion.

The qubit reference type $\texttt{qref}\langle q\rangle$ is a singleton type [3, 20], which is equivalent to $ptr(l)$ in alias types [53]. Qubit symbols are shown in orange to distinguish them from the usual variables denoted by the metavariable x; we use qubit symbols to model the underlying logical qubit that the surface Q# language does not expose. Unitary operations, U (shown in *pink*), are parametric to the grammar; similar to Q#, which does not prefer a specific gate set. An n-qubit unitary is typed as $U : \times_{i\in n}(i \hookrightarrow \texttt{qref}\langle q_i\rangle) \to \texttt{cmd}(\texttt{unit})$, where $\dim(U) = 2^n$. This type ensures that multi-qubit gates can be applied only to distinct qubits. The $\texttt{apply}_U(e)$ command applies the given unitary operation to a tuple of unique qubit references, where we follow singleton-tuple equivalence like Q# in case of a single-qubit unitary. Controlled unitaries can be represented using block diagonals, e.g., $CNOT \triangleq \mathrm{D}_{(I_2,X)}$ and are typed as $\mathrm{D}_{(U,V)} : \times_{i\in n+1}(i \hookrightarrow \texttt{qref}\langle q_i\rangle) \to \texttt{cmd}(\texttt{unit})$, where $\dim(U) = \dim(V) = 2^n$. It is understood that

Sort			*Abstract*	*Concrete*	
Typ	τ	$::=$	$\mathtt{qref}\langle q\rangle$	$\mathtt{qref}\langle q\rangle$	qubit reference
			$\mathtt{fun}(\tau_1;\tau_2)$	$\tau_1 \to \tau_2$	function
			$\mathtt{cmd}(\tau)$	$\mathtt{cmd}(\tau)$	command
			$\mathtt{prod}(i \hookrightarrow \tau_i \mid i \in n)$	$\times_{i\in n}(i \hookrightarrow \tau_i)$	variadic product
			$\mathtt{bool}$	$\mathtt{bool}$	boolean
			$\mathtt{unit}$	$\mathtt{unit}$	unit
Exp	e	$::=$	x	x	variable
			$\mathtt{let}[\tau_1;\tau_2](e_1;x.e_2)$	$\mathtt{let}\ x\ \mathtt{be}\ e_1\ \mathtt{in}\ e_2$	let binding
			$\lambda[\tau_1;\tau_2](x.e)$	$\lambda(x.e)$	function
			$\mathtt{ap}[\tau_1;\tau_2](e_1;e_2)$	$\mathtt{ap}(e_1;e_2)$	application
			$\mathtt{cmd}[\tau](m)$	$\mathtt{cmd}(m)$	encapsulated command
			$\mathtt{tuple}[\tau_n](i \hookrightarrow e_i \mid i \in n)$	$\langle i \hookrightarrow e_i \mid i \in n\rangle$	tuple
			$\mathtt{proj}\langle i\rangle[\tau_n](e)$	$e \cdot i$	projection
			$\mathtt{true}$	$\mathtt{true}$	true
			$\mathtt{false}$	$\mathtt{false}$	false
			$\mathtt{if}[\tau](e;e_1;e_2)$	$\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$	if expression
			$\mathtt{unit}$	$\langle\rangle$	unit
Cmd	m	$::=$	$\mathtt{ret}[\tau](e)$	$\mathtt{ret}(e)$	return
			$\mathtt{bnd}[\tau_1;\tau_2](e;x.m)$	$\mathtt{bnd}\ x \leftarrow e\ ;\ m$	bind
			$\mathtt{newqref}[\tau](x.m)$	$\mathtt{new}\ x\ \mathtt{in}\ m$	new qubit reference
			$\mathtt{gateap}\langle U_{2^n}\rangle(e)$	$\mathtt{apply}_U(e)$	gate application
			$\mathtt{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1;e_2)$	$\mathtt{apply}_{D(U,V)}(e_1;e_2)$	diagonal gate application
			$\mathtt{meas}(e)$	$\mathtt{meas}(e)$	measure

Figure 2: Abstract and concrete syntax of $\lambda_{Q\#}$.

the number of arguments required for both forms of gate application depends on the dimension of the unitary parameters involved and is enforced by the typing rules.

3.2 Static Semantics

The pure fragment of $\lambda_{Q\#}$ is the usual simply-typed λ-calculus, so we will not say much about it here. We show typing rules for the effectful portion of $\lambda_{Q\#}$ in Figure 3.

All of our command typing judgments are parameterized by a signature, Σ, that keeps track of available qubit symbols in scope and corresponds to the shape of the quantum memory, much like store shapes[3] in the semantics of ALGOL [36, 37, 48]. The intuition behind incorporating a signature is that the block structure induced by the allocation command changes the shape of the quantum memory under consideration by making a new qubit available to the program on entry and removing it on exit. This is the essence of the stack-like treatment of local state.[4]

Rules CMD-RET and CMD-BND are the two standard rules for monadic return and bind operations. The other four rules are specific to quantum computation.

The $\mathtt{newqref}(x.m)$ command allocates a fresh logical qubit q and immediately makes a reference to it available in the scope of m. Its typing rule CMD-NEWQREF says that if command m returns a value of

[3] Store shapes follow laws similar to what are known as lenses in the current literature [9].

[4] Another view is to think of the commands as being parametrically polymorphic [33, 34] to the store, an idea considered by Reynolds as early as 1975 [6, 47], even before store shapes. Still, we prefer the signature-based approach taken in Harper's **MA**.

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

$$\textsc{cmd-Ret}\ \frac{\Gamma \vdash_\Sigma e : \tau}{\Gamma \vdash_\Sigma \mathtt{ret}(e) \div \tau} \qquad \textsc{cmd-Bnd}\ \frac{\Gamma \vdash_\Sigma e : \mathtt{cmd}(\tau) \quad \Gamma, x:\tau \vdash_\Sigma m \div \tau'}{\Gamma \vdash_\Sigma \mathtt{bnd}(e; x.m) \div \tau'} \qquad \textsc{cmd-NewQRef}\ \frac{\Gamma, x : \mathtt{qref}\langle q\rangle \vdash_{\Sigma,q} m \div \tau}{\Gamma \vdash_\Sigma \mathtt{newqref}(x.m) \div \tau}$$

$$\textsc{cmd-GateApRef}\ \frac{\Gamma \vdash_\Sigma e : \mathtt{prod}(\overline{i \hookrightarrow \mathtt{qref}\langle q_i\rangle}^{\,i\in 1..n})}{\Gamma \vdash_\Sigma \mathtt{gateap}\langle U_{2^n}\rangle(e) \div \mathtt{unit}} \qquad \textsc{cmd-DiagApRef}\ \frac{\Gamma \vdash_\Sigma e_1 : \mathtt{qref}\langle q\rangle \quad \Gamma \vdash_\Sigma e_2 : \mathtt{prod}(\overline{i \hookrightarrow \mathtt{qref}\langle r_i\rangle}^{\,i\in 1..n})}{\Gamma \vdash_\Sigma \mathtt{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1; e_2) \div \mathtt{unit}}$$

$$\textsc{cmd-MeasRef}\ \frac{\Gamma \vdash_\Sigma e : \mathtt{qref}\langle q\rangle}{\Gamma \vdash_\Sigma \mathtt{meas}(e) \div \mathtt{bool}}$$

Figure 3: Typing of commands. Γ is the standard typing context, and the signature, Σ, keeps track of qubit symbols in scope. Each qubit symbol is required to be distinct. See other rules in Appendix B.1.

type τ in a context containing $x : \mathtt{qref}\langle q\rangle$ and a signature extended with q, then $\mathtt{newqref}(x.m)$ returns a value of type τ. The binding structure ensures that the lifetime of the newly allocated qubit is equal to its lexical scope, ensuring a strict stack discipline and providing safe and automatic management of qubits.

Rules CMD-GateApRef and CMD-DiagApRef for unitary operations enforce the constraint that the input qubit references are distinct. Rule CMD-MeasRef shows how to obtain a boolean value from an expression that resolves to a qubit reference.

In summary, the allocation command changes the shape of the store (quantum state under consideration), while commands like unitary application and measurement change the store (quantum state).

3.2.1 Safety Properties

We claim that our type system supports two safety properties currently not offered by Q#:

Proposition 1. *$\lambda_{Q\#}$ supports controlled aliasing and hence, statically enforces the no-cloning theorem for all unitary operations.*

This follows from rules CMD-GateApRef and CMD-DiagApRef: The premises of both typing rules require the input qubit references to be unique as all the entries in a tuple are required to be references to different logical qubits. In the case of the block diagonal, the control qubit reference, e_1, is also required to be distinct from the qubit references in e_2.

Example 3.1. The unsafe code fragment from Figure 1b can be written in $\lambda_{Q\#}$ syntax as:

$$\mathtt{newqref}(q_1.\mathtt{ret}(\mathtt{let}(q_1; q_2.\mathtt{cmd}(\mathtt{diagap}\langle I_2, X_2\rangle(q_1; q_2)))))$$

or in the concrete syntax as $\mathtt{new}\ q_1\ \mathtt{in}\ \mathtt{ret}(\mathtt{let}\ q_2\ \mathtt{be}\ q_1\ \mathtt{in}\ \mathtt{cmd}(\mathtt{apply}_{D(I_2,X)}(q_1; q_2)))$. Since the type of the qubit reference in $\lambda_{Q\#}$, $\mathtt{qref}\langle q\rangle$, is indexed by the symbolic name of the qubit, we can tell statically that q_1 and q_2 reference the same underlying logical qubit. This allows our type system to reject the above program even though the Q# compiler allows it to pass.

Proposition 2. *$\lambda_{Q\#}$ statically ensures safe memory management and disallows dangling qubit references.*

The allocation command, $\mathtt{newqref}(x.m)$, as previously explained, comes with its own binding form, which ensures that the reference created during allocation can never escape its lexical scope. In rule CMD-NewQRef, the fresh logical qubit q allocated during this command is only available in the extended signature in the premise and not in the conclusion at the end of the command.

Example 3.2. Using $\lambda_{Q\#}$ concrete syntax and the derived forms from Appendix A, the unsafe code fragment shown in Figure 1a can be written as $\texttt{let}\ \text{NewQubit}\ \texttt{be}\ \texttt{proc}()\ \{\ \texttt{new}\ x\ \texttt{in}\ \texttt{ret}(x)\ \}\ \texttt{in}\ \langle\rangle$. Here, while $\texttt{ret}(x)$ has type $\texttt{qref}\langle q\rangle$ when q is in the signature, i.e., $\Gamma, x : \texttt{qref}\langle q\rangle \vdash_{\Sigma,q} \texttt{ret}(x) \dot{\sim} \texttt{qref}\langle q\rangle$; the conclusion of rule CMD-NEWQREF removes q from scope and renders $\texttt{new}\ x\ \texttt{in}\ \texttt{ret}(x)$ ill-typed, i.e., $\Gamma \nvdash_{\Sigma} \texttt{new}\ x\ \texttt{in}\ \texttt{ret}(x) \dot{\sim} \texttt{qref}\langle q\rangle$.

3.3 Dynamic Semantics

As Staton [55, p. 3] suggests, "by giving a fully complete equational theory we can understand quantum computation from the axioms of the theory without having to turn to denotational models built from operator algebra"; we rely on his equational theory for quantum local store [55, §6.2] to provide an equational dynamics for the effectful quantum fragment of $\lambda_{Q\#}$.[5] Unlike Staton's language, our unitary operations do not return qubits but modify them in place. In this presentation, we use several derived forms from Appendix A. Specifically, `do` returns the result of sequential execution of commands. The program equations assume the availability of a universal gate set.

Interesting Axioms

$$a : \texttt{qref}\langle q\rangle \vdash \texttt{do}\ \{\texttt{apply}_X(a); \texttt{meas}(a)\} \equiv \neg\ \texttt{do}\ \{\texttt{meas}(a)\} \tag{A}$$

$$a : \texttt{qref}\langle q\rangle,\ b : \bigtimes_{i\in n}(i \hookrightarrow \texttt{qref}\langle r_i\rangle) \vdash \{\texttt{apply}_{D(U,V)}(a;b); \texttt{meas}(a); \texttt{ret}(\langle\rangle)\} \equiv \{x \leftarrow \texttt{meas}(a); \texttt{ret}(\texttt{if}\ x\ \texttt{then}\ \texttt{cmd}(\texttt{apply}_V(b))\ \texttt{else}\ \texttt{cmd}(\texttt{apply}_U(b)))\} \tag{B}$$

$$\cdot \vdash \texttt{do}\ \{\texttt{new}\ a\ \texttt{in}\ \texttt{meas}(a)\} \equiv \texttt{false} \tag{D}$$

$$b : \texttt{qref}\langle q\rangle \vdash \texttt{do}\ \{\texttt{new}\ a\ \texttt{in}\ \texttt{apply}_{D(U,V)}(a;b)\} \equiv \texttt{do}\ \{\texttt{apply}_U(b); \texttt{new}\ a\ \texttt{in}\ \texttt{ret}(\langle\rangle)\} \tag{E}$$

As mentioned in §2.2, we omit Staton's axiom (C) because, in the QRAM model, the discard operation just forgets the name of the reference to a qubit. In Q# and $\lambda_{Q\#}$, we may consider an equivalent behavior: qubit references are automatically forgotten when they reach the end of their lexical scope. A degenerate case of axiom (C) (for a 1×1 unitary) holds for both Staton's theory for a quantum local store and $\lambda_{Q\#}$; it says that one can ignore the global phase.

Administrative Axioms The following equations correspond to respecting the composition and product monoidal structure of unitaries:

$$m_1 : \texttt{cmd}(\tau_1),\ m_2 : \texttt{cmd}(\tau_2) \vdash \texttt{do}\ \{\texttt{new}\ a\ \texttt{in}\ \texttt{new}\ b\ \texttt{in}\ m_1; \texttt{apply}_{SWAP}(a,b); m_2\} \equiv \texttt{do}\ \{\texttt{new}\ a\ \texttt{in}\ \texttt{new}\ b\ \texttt{in}\ m_1; \texttt{let}\ \langle b,a\rangle\ \texttt{be}\ \langle a,b\rangle\ \texttt{in}\ \texttt{cmd}(m_2)\} \tag{F}$$

$$e : \bigtimes_{i\in n}(i \hookrightarrow \texttt{qref}\langle q_i\rangle) \vdash \texttt{do}\ \{\texttt{apply}_{I_{2^n}}(e)\} \equiv \langle\rangle \tag{G}$$

$$e : \bigtimes_{i\in n}(i \hookrightarrow \texttt{qref}\langle q_i\rangle) \vdash \texttt{do}\ \{\texttt{apply}_{VU}(e)\} \equiv \texttt{do}\ \{\texttt{apply}_U(e); \texttt{apply}_V(e)\} \tag{H}$$

$$e_1 : \bigtimes_{i\in m}(i \hookrightarrow \texttt{qref}\langle q_i\rangle),\ e_2 : \bigtimes_{i\in n}(i \hookrightarrow \texttt{qref}\langle r_i\rangle) \vdash \texttt{do}\ \{\texttt{apply}_{U\otimes V}(e_1,e_2)\} \equiv \texttt{do}\ \{\texttt{apply}_U(e_1); \texttt{apply}_V(e_2)\} \tag{I}$$

Selinger [50] notes that the *SWAP* gate is equivalent to classically renaming qubit references, which captures the intuition behind equation (F). However, in our case, we have to ensure that the scope of the

[5] We show the traditional operational semantics in Appendix B.2.

qubits is limited to our expression (since *SWAP* is stateful). Axiom (G) says that applying an identity gate is equivalent to doing nothing. Axioms (H) and (I) show the two ways of composing unitaries—sequential and tensor products (horizontal and vertical composition, respectively, in circuit notation).

Like Staton, we also state the commutativity equations that hold for $\lambda_{Q\#}$:

$$a : \texttt{qref}\langle q\rangle,\ b : \texttt{qref}\langle r\rangle,\ m : \texttt{cmd}(\tau) \vdash \texttt{do}\ \{x \leftarrow \texttt{meas}(a)\ ;\ y \leftarrow \texttt{meas}(b)\ ;\ m\} \equiv \texttt{do}\ \{y \leftarrow \texttt{meas}(b)\ ;\ x \leftarrow \texttt{meas}(a)\ ;\ m\} \quad \text{(J)}$$

$$m : \texttt{cmd}(\tau) \vdash \texttt{do}\ \{\texttt{new}\ a\ \texttt{in}\ \texttt{new}\ b\ \texttt{in}\ m\} \equiv \texttt{do}\ \{\texttt{new}\ b\ \texttt{in}\ \texttt{new}\ a\ \texttt{in}\ m\} \quad \text{(K)}$$

$$b : \texttt{qref}\langle q\rangle,\ m : \texttt{cmd}(\tau) \vdash \texttt{do}\ \{\texttt{new}\ a\ \texttt{in}\ y \leftarrow \texttt{meas}(b)\ ;\ m\} \equiv \texttt{do}\ \{y \leftarrow \texttt{meas}(b)\ ;\ \texttt{new}\ a\ \texttt{in}\ m\} \quad \text{(L)}$$

Now that we have shown that the quantum portion of $\lambda_{Q\#}$ is equivalent to Staton's quantum programming language [55, §5][6] and corresponding program equations with his theory of quantum local store, we can restate Staton's result [55, p. 8, Theorem 11] for our language:

Theorem 1 (Universality of $\lambda_{Q\#}$)**.** *For any linear map $f : M_{2^{n_1}} \oplus \cdots \oplus M_{2^{n_k}} \to M_{2^p}$ that is completely positive and unital (i.e. corresponds to a trace-preserving superoperator), there is a $\lambda_{Q\#}$ term, t, such that t implements f.*

This corresponds to Staton's Theorem 11.1 [55], which is a variation on Selinger's Theorem 6.14 [50]. The proof relies on the correspondence between Staton's simple quantum language and a fragment of $\lambda_{Q\#}$, where the translation is straightforward (Appendix C).

Theorem 2 (Completeness)**.** *Assuming an axiomatization of unitaries, if two terms t and u have equivalent interpretation in a common context, Γ, then $\Gamma \vdash t \equiv u$ is derivable.*

Note that since Q# is parameterized over gate sets, we need an equational theory over unitaries for $\lambda_{Q\#}$. In the simplest case, we can declare two unitaries equal if their corresponding matrices are equal. Again, the result follows from Staton's Theorem 11.2. There are two differences: (1) instead of algebraic operations, our theorem is stated in terms of generic effects (which correspond directly to programming); (2) in addition to the equations (A)–(L) stated above, we also need the standard $\beta\eta$-equalities of simply-typed λ-calculus, which are required because our axioms do not live in isolation but are written as typed expressions in a context.

4 Translation from Q# to $\lambda_{Q\#}$

We summarize the rules for converting from the supported features of Q# to $\lambda_{Q\#}$ in Table 1. For ease of presentation, we use the derived forms from Appendix A. Figure 4 in Appendix D shows the elaboration of the Q# teleport example from Listing 1.

Elaboration maintains a *context* (not shown in Table 1) that stores the logical qubit associated with each qubit reference. To translate the Q# type `Qubit` to the $\lambda_{Q\#}$ type $\texttt{qref}\langle q\rangle$, we look up the reference associated with the `Qubit` type in the context or add a new logical qubit to the context. The `use` statements and `operation` parameters update the context to include a mapping from the new qubit or qubit parameter(s) to a fresh logical qubit. In Figure 4, a, b, and m are distinct logical qubits introduced by elaboration.

Elaboration performs some type checking to produce well-formed $\lambda_{Q\#}$ terms. For example, elaboration checks that the first argument to a `Controlled` or `Adjoint` functor is equipped with the `Ctl` and/or `Adj`

[6] See Appendix C for the trivial term translation.

Q# Syntax	$\lambda_{Q\#}$ Translation
$[\![(\tau_1, \ldots, \tau_n)]\!]$	$\times_{i \in n}(i \hookrightarrow [\![\tau_i]\!])$
$[\![\tau_1 \texttt{ -> } \tau_2]\!]$	$[\![\tau_1]\!] \to [\![\tau_2]\!]$
$[\![\tau_1 \texttt{ => } \tau_2]\!]$	$[\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$
$[\![\texttt{Bool}]\!]$ and $[\![\texttt{Result}]\!]$	`bool`
$[\![\texttt{Qubit}]\!]$	$\texttt{qref}\langle q \rangle$
$[\![\texttt{Unit}]\!]$	`unit`
$[\![\texttt{function } f\ (x_1 : \tau_1, \ldots) : \tau\ \{\ e\ \}]\!]$	$\lambda[\times_{i \in n}(i \hookrightarrow [\![\tau_i]\!]);\ \ \tau](\langle i \hookrightarrow x_i \mid i \in n \rangle\,.\,[\![e]\!])$
$[\![\texttt{operation } f\ (x_1 : \tau_1, \ldots) : \tau\ \{\ s\ \}]\!]$	$\lambda[\times_{i \in n}(i \hookrightarrow [\![\tau_i]\!]);\ \texttt{cmd}(\tau)](\langle i \hookrightarrow x_i \mid i \in n \rangle\,.\,[\![s]\!])$
$[\![\texttt{return } e]\!]$	$\texttt{ret}([\![e]\!])$
$[\![\texttt{let } x = e; \ldots]\!]$	$\texttt{let } x \texttt{ be } [\![e]\!] \texttt{ in } [\![\ldots]\!]$
$[\![\texttt{if } e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}]\!]$	$\texttt{if } [\![e]\!] \texttt{ then cmd}([\![s_1]\!]) \texttt{ else cmd}([\![s_2]\!])$
$[\![\texttt{use } q = \texttt{Qubit}\ \{\ s\ \}]\!]$	$\texttt{new } q \texttt{ in } [\![s]\!]$
$[\![\texttt{Adjoint } e_1\ (e_2)]\!]$	$\texttt{apply}_{U^\dagger}([\![e_2]\!])$, where $U = \text{mat}([\![e_1]\!])$
$[\![\texttt{Controlled } e_1\ (q, e_2)]\!]$	$\texttt{apply}_{D(I_{2^n}, U)}(q\,;[\![e_2]\!])$, where $U = \text{mat}([\![e_1]\!])$
$[\![e_1\ (e_2)]\!]$	$[\![e_1]\!]([\![e_2]\!])$
$[\![(e_1, \ldots, e_n)]\!]$	$\langle [\![e_i]\!], \ldots, [\![e_n]\!] \rangle$
$[\![\texttt{true}]\!]$ and $[\![\texttt{One}]\!]$	`true`
$[\![\texttt{false}]\!]$ and $[\![\texttt{Zero}]\!]$	`false`

Table 1: Select Q# to $\lambda_{Q\#}$ elaboration rules. f, x, and q are variable names, e is a Q# expression, s is a Q# statement, and τ is a Q# type. $[\![\cdot]\!]$ is the elaboration function and $\text{mat}(\cdot)$ converts a $\lambda_{Q\#}$ expression to its corresponding unitary operator. In the rule for `Qubit`, q is determined from the elaboration context.

characteristics and inlines the corresponding operation specialization, converting it to a unitary operator. We need to do this during elaboration because we do not yet encode characteristic information or specializations in $\lambda_{Q\#}$. An adjointable Q# operation with type `(Qubit,...,Qubit) => Unit` can be converted into a unitary matrix by composing the unitary representations of its primitive gates. Note that the type signature and the fact that the operation is adjointable (i.e., no measurement) mean it can be unfolded to a sequence of primitive gates. We also expand multi-controlled operations (`Controlled` statements with a list of controls) into a nested group of single-qubit controlled operations, and expand `if-elif-else` expressions into nested if-then-else expressions using $\langle\rangle$ in place of an empty `else` block. Finally, we restrict Q# `function` bodies to be pure expressions since we do not handle classical `mutable` values.

As this is our initial attempt to get the foundations right, we do not yet support several Q# features: namespaces; operation characteristics; custom operation specializations (i.e., implementations of controlled or adjoint variants); general application of the `Adjoint` and `Controlled` functors; arrays and slices; type parameters; base types outside of `Bool`, `Result`, and `Unit`; iteration using `for`, `while`, or `repeat`; and `within-apply` blocks (which apply an operation and its adjoint). We say more in §6 about the challenges involved in supporting some of these features.

5 Related Work

Large Language Definition Efforts In starting this project, we were encouraged by previous efforts in the formal specification of large programming languages such as Standard ML [19, 28], Java [23], JavaScript [15], Rust [24, 25], and, most recently, Go [13]. As we mentioned in §3, we more or less

followed the pioneering methodology of the formalization and mechanization of the definition of Standard ML [29] by identifying a well-founded core language and performing all metatheoretical reasoning on that core. These projects demonstrate the extent to which it is possible to distill large and complex languages into their formal and faithful essence. Java and Go serve as examples of industry-scale languages in mass use benefiting from formalization and academic study: Extensions such as generics (polymorphism) were first investigated on smaller cores of the respective languages before being adopted in production over the years. In the case of JavaScript, perhaps the impact of a careful formal study was even more significant as the language became the assembly of the web. We hope our work serves as a similar playground for extensions and future impact.

Equational Theories Like Staton, we do not focus on the axiomatization of unitaries but of quantum computation in general. We discuss two similar works here.

Paykin and Zdancewic [41] build upon Staton's work and present an equational theory for quantum computation embedded inside homotopy type theory (HoTT) [58]. The essential idea to treat unitaries as higher inductive paths simplifies the presentation of the equational theory as several axioms can be derived using the rich structure of HoTT. While their work focuses on embedding a quantum language inside a highly expressive dependent type theory, we are motivated by practical concerns in defining semantics for a real-world quantum language.

Peng et al. [42] introduce Non-Idempotent Kleene Algebra (NKAT) to reason about programs algebraically. Their language is based on Kozen's Kleene Algebra with Test (or KAT), which models both programs and assertions, allowing for a lightweight implementation of a Hoare-style logic [27]. While the underlying language of regular expressions is not designed for convenience in programming, their use of NKAT to verify quantum program transformations is a key use-case of equational theories and one we plan to explore in future.

Linearity Research-oriented languages like $\mathcal{Q}$WIRE [40] and Silq [5] employ a linear type system to enforce the no-cloning theorem. So far, industry languages, including Q#, have not adopted linear typing. The lack of linear typing in Q# is justified by its monadic treatment of state. That is, the monad interface imposes a sequential order to manipulate the quantum state as every monad can be treated as a linear-use state monad [30]. The design decision in Q# to permit uncontrolled aliasing of qubits for user comfort is the only reason a monadic interface is not enough, which is addressed by our type system.

ALGOL-like Quantum Languages The IQu language [39] extends Idealized ALGOL with quantum circuits and quantum variables, much as we extend Harper's Modernized Algol (MA). Like $\lambda_{Q\#}$, IQu uses references to access qubits and therefore does not need a linear type system to prevent cloning. Though every newly allocated qubit is unique, IQu does not have a way to guarantee that multiple references to the same qubit are not passed to a single operation. Instead, IQu's use of Idealized ALGOL is focused on programmability, following a design philosophy similar to that of Q#. IQu allows programmers to write the classical parts of their programs in a familiar way while providing access to the quantum state.

6 Conclusion and Perspectives

We present a core calculus for the Q# programming language, dubbed $\lambda_{Q\#}$. We maintain a separation between the quantum effectful and the pure expression sub-languages, expose the monadic nature of

computation inherent in Q#, make qubit aliasing and block structure explicit, and present an equational semantics for $\lambda_{Q\#}$, building upon Staton's fully complete equational theory for quantum computation.

A formal specification of the whole Q# language still requires more work. Some extensions are straightforward; e.g., classical mutable bindings in Q# can be modeled after assignables in Harper's **MA**; conveniently, they follow the model of classical local store analogous to how we modeled the quantum local store in this paper. Here it helps that Q# does not allow references to any types other than qubits. Other features are more challenging, including arrays, slices, iteration, polymorphism, and patterns like `within-apply` and repeat-until-success [38]. Then there is the question of how to treat operations that have specializations supplied by the programmer versus those auto-generated by the Q# compiler (which is not known statically); we may need to consider a phase distinction [18] here to distinguish between what can be derived statically using types and what requires inspecting the code.

We plan to gain confidence in our formalization by mechanizing its metatheory. We see potential in recent developments such as the Agda-based formalization of Second-Order Abstract Syntax [8], which lets users concisely specify algebraic theories such as Staton's and significantly reduces the boilerplate code required to state interesting theorems about the theory. However, this tool does not support substructural assumptions on qubit symbols, making our proposed extension a nontrivial prospect.

A major goal of this project is to form a playground for prototyping extensions to the Q# type system. For instance, a peculiar decision in Q# is to allow uncontrolled aliasing of qubits in support of user-friendly features such as qubit arrays. While convenient, reasoning about interference freedom for arrays is notoriously hard; specifically, our approach to enforce no-cloning inspired by alias types [53, 60] does not easily scale to arrays [59, §3.5.1]. We are extending our $\lambda_{Q\#}$ type checker with a constraint solver to evaluate potential solutions for scenarios that occur in practice in Q# library code. Depending on the complexity of the array indexing used in practice, we may use a natural number inequality checker, a simple symbolic numerical solver, or a full-fledged SMT solver like Z3 [31] to guarantee qubit distinctness.

We could also statically check Q#'s `Adj` and `Ctl` characteristics for validity. In the simplest case, we would flag operations as unitary or non-unitary in order to inform the compiler when adjoints and controls can be trivially synthesized, in the manner of Silq [5]. However, more complex programs are adjointable and controllable in practice, which may require a more sophisticated approach.

One of our insights from this project is that even though quantum computation is a fundamentally new abstraction, many classical techniques both from programming languages and compilers communities can be adapted to the quantum setting [56]. Q# and its Quantum Development Kit (QDK) are significant examples of realizing that vision [1]. But as a high-level programming language, Q# must also compile to efficient, low-level machine instructions. Recently, Microsoft announced QIR, a Quantum Intermediate Representation based on the popular LLVM framework [12], which has gained significant industry backing in the form of the QIR Alliance [45]. This provides an exciting avenue for future development. We plan to explore semantics-preserving compilation from Q# to QIR using our formalization. This project will require formally specifying the semantics of QIR, for which we will draw upon the Verified LLVM (Vellvm) project [62]. We also aim to formalize QIR's *profiles*, which specify what kinds of quantum operations are allowed on a given quantum architecture. This, along with our current work, will constitute a significant step toward our broader vision of a fully verified quantum stack [46].

References

[1] Alfred Aho & Jeffrey Ullman (2022): *Abstractions, Their Algorithms, and Their Compilers*. *Commun. ACM* 65(2), pp. 76–91, doi:10.1145/3490685.

[2] Thorsten Altenkirch & Alexander S. Green (2009): *The Quantum IO Monad*. In: *Semantic Techniques in Quantum Computation*, Cambridge University Press, pp. 173–205, doi:10.1017/CBO9781139193313.006. Available at https://www.cs.nott.ac.uk/~psztxa/g5xnsc/chapter.pdf.

[3] David Aspinall (1995): *Subtyping with Singleton Types*. In Leszek Pacholski & Jerzy Tiuryn, editors: *Computer Science Logic*, *Lecture Notes in Computer Science* 933, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–15, doi:10.1007/BFb0022243. Available at https://homepages.inf.ed.ac.uk/da/papers/lss/.

[4] John Azariah (2018): *F# & Q# - A tale of two languages*. F# and Q# Advent Calendars 2018. Available at https://johnazariah.github.io/2018/12/04/tale-of-two-languages.html.

[5] Benjamin Bichsel, Maximilian Baader, Timon Gehr & Martin Vechev (2020): *Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics*. In: *Proc. PLDI '20*, ACM, pp. 286–300, doi:10.1145/3385412.3386007. Available at https://files.sri.inf.ethz.ch/website/papers/pldi20-silq.pdf.

[6] Stephen Brookes, Peter W. O'Hearn & Uday Reddy (2014): *The Essence of Reynolds*. In: *Proc. POPL '14*, ACM, pp. 251–255, doi:10.1145/2535838.2537851.

[7] Karl Crary & Robert Harper (2009): *Mechanized Definition of Standard ML*. Available at https://github.com/SMLFamily/The-Mechanization-of-Standard-ML.

[8] Marcelo Fiore & Dmitrij Szamozvancev (2022): *Formal Metatheory of Second-Order Abstract Syntax*. *Proc. ACM Program. Lang.* 6(POPL), doi:10.1145/3498715. Source: https://github.com/DimaSamoz/agda-soas.

[9] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce & Alan Schmitt (2007): *Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem*. *ACM Trans. Program. Lang. Syst.* 29(3), doi:10.1145/1232420.1232424.

[10] Alan Geller (2018): *Qubits in Q#*. Q# Advent Calendar 2018. Available at https://devblogs.microsoft.com/qsharp/qubits-in-qsharp/.

[11] Alan Geller (2019): *What are Qubits?* Q# Advent Calendar 2019. Available at https://devblogs.microsoft.com/qsharp/what-are-qubits/.

[12] Alan Geller (2020): *Introducing QIR*. Q# Blog. Available at https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/.

[13] Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler & Nobuko Yoshida (2020): *Featherweight Go*. *Proc. ACM Program. Lang.* 4(OOPSLA), doi:10.1145/3428217.

[14] Lov K. Grover (1996): *A Fast Quantum Mechanical Algorithm for Database Search*. In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, ACM, New York, NY, USA, pp. 212–219, doi:10.1145/237814.237866. arXiv:quant-ph/9605043.

[15] Arjun Guha, Claudiu Saftoiu & Shriram Krishnamurthi (2010): *The Essence of JavaScript*. In: *ECOOP 2010 – Object-Oriented Programming*, Springer, pp. 126–150, doi:10.1007/978-3-642-14107-2_7. arXiv:1510.00925.

[16] Robert Harper (2016): *Practical Foundations for Programming Languages*, 2nd edition, chapter 34: Modernized Algol, pp. 301–312. Cambridge University Press, doi:10.1017/CBO9781316576892.036.

[17] Robert Harper (2016): *Practical Foundations for Programming Languages*, 2nd edition. Cambridge University Press, doi:10.1017/CBO9781316576892. Abbreviated online edition, with corrections: https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf.

[18] Robert Harper (2021): *Phase Distinctions in Type Theory*. Talk at The Topos Institute Colloquium. Available at https://www.cs.cmu.edu/~rwh/talks/ti-talk.pdf.

[19] Robert Harper & Christopher A. Stone (2000): *A Type-Theoretic Interpretation of Standard ML*. In: *Proof, Language, and Interaction: Essays in Honor of Robin Milner*, MIT Press, pp. 341–387. Available at https://www.cs.cmu.edu/~rwh/papers/ttisml/ttisml.pdf.

[20] Susumu Hayashi (1994): *Singleton, Union, and Intersection Types for Program Extraction*. *Information and Computation* 109(1–2), pp. 174–210, doi:10.1006/inco.1994.1016.

[21] Bettina Heim (2020): *Development of Quantum Applications*. Ph.D. thesis, ETH Zurich, doi:10.3929/ethz-b-000468201. Ch. 8: Domain-Specific Language Q#.

[22] Bettina Heim & Q# Team (2020): *Q# Language Specification*. Available at https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language. Also see [21].

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

[24] Ralf Jung (2020): *Understanding and Evolving the Rust Programming Language*. Ph.D. thesis, Saarland University, doi:10.22028/D291-31946. Available at https://www.ralfj.de/research/thesis.html.

[25] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers & Derek Dreyer (2017): *RustBelt: Securing the Foundations of the Rust Programming Language*. *Proc. ACM Program. Lang.* 2(POPL), doi:10.1145/3158154.

[26] Emmanuel Knill (1996): *Conventions for Quantum Pseudocode*. Technical Report LA-UR-96-2724, Los Alamos National Laboratory, doi:10.2172/366453.

[27] Dexter Kozen (1997): *Kleene Algebra with Tests*. *ACM Trans. Program. Lang. Syst.* 19(3), pp. 427–443, doi:10.1145/256167.256195.

[28] Daniel K. Lee, Karl Crary & Robert Harper (2007): *Towards a Mechanized Metatheory of Standard ML*. In: *Proc. POPL '07*, ACM, pp. 173–184, doi:10.1145/1190216.1190245. Available at https://www.cs.cmu.edu/~dklee/papers/tslf-popl.pdf.

[29] Robin Milner, Mads Tofte & David Macqueen (1997): *The Definition of Standard ML (Revised)*. MIT Press. Available at https://smlfamily.github.io/sml97-defn.pdf.

[30] Rasmus Ejlers Møgelberg & Sam Staton (2014): *Linear Usage of State*. *Log. Methods Comput. Sci.* 10(1), doi:10.2168/LMCS-10(1:17)2014.

[31] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*, Springer, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[32] Mariia Mykhailova (2020): *The Quantum Katas: Learning Quantum Computing Using Programming Exercises*. In: *Proc. SIGCSE 2020*, ACM, p. 1417, doi:10.1145/3328778.3372543. Source: https://github.com/microsoft/QuantumKatas.

[33] Peter W. O'Hearn & John C. Reynolds (2000): *From Algol to Polymorphic Linear Lambda-Calculus*. *J. ACM* 47(1), pp. 167–223, doi:10.1145/331605.331611.

[34] Peter W. O'Hearn & Robert D. Tennent (1995): *Parametricity and Local Variables*. *J. ACM* 42(3), pp. 658–709, doi:10.1145/210346.210425. Reprinted as [35].

[35] Peter W. O'Hearn & Robert D. Tennent (1997): *Parametricity and Local Variables*. In: *ALGOL-like Languages (Volume 2)*, Birkhäuser Boston, pp. 109–163, doi:10.1007/978-1-4757-3851-3_6.

[36] Frank J. Oles (1982): *A Category-Theoretic Approach to the Semantics of Programming Languages*. Ph.D. thesis, Syracuse University. Available at https://www.cs.cmu.edu/afs/cs.cmu.edu/project/fox-19/member/jcr/www/FrankOlesThesis.pdf. Order no. AAI8301650.

[37] Frank J. Oles (1985): *Type Algebras, Functor Categories and Block Structure*. In: *Algebraic Methods in Semantics*, Cambridge University Press, pp. 543–573, doi:10.7146/dpb.v12i156.7430.

[38] Adam Paetznick & Krysta M. Svore (2014): *Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries*. *Quantum Inf. Comput.* 14(15 & 16), pp. 1277–1301, doi:10.26421/qic14.15-16-2. Corresponding Q# code sample: https://docs.microsoft.com/en-us/samples/microsoft/quantum/repeat-until-success/.

[39] Luca Paolini, Luca Roversi & Margherita Zorzi (2019): *Quantum Programming Made Easy*. In: *Proc. Linearity-TLLA '18*, *EPTCS* 292, pp. 133–147, doi:10.4204/EPTCS.292.8.

[40] Jennifer Paykin, Robert Rand & Steve Zdancewic (2017): *QWIRE: A Core Language for Quantum Circuits*. In: *Proc. POPL '17*, ACM, pp. 846–858, doi:10.1145/3009837.3009894. Available at https://jpaykin.github.io/papers/prz_qwire_2017.pdf.

[41] Jennifer Paykin & Steve Zdancewic (2019): *A HoTT Quantum Equational Theory (Extended Version)*. QPL '19. arXiv:1904.04371.

[42] Yuxiang Peng, Mingsheng Ying & Xiaodi Wu (2022): *Algebraic Reasoning of Quantum Programs via Non-Idempotent Kleene Algebra*. In: *Proc. 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '22, ACM, New York, NY, USA, p. 14, doi:10.1145/3519939.3523713. arXiv:2110.07018. To appear.

[43] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf—A Meta-Logical Framework for Deductive Systems*. In: *Automated Deduction—CADE-16*, Springer, pp. 202–206, doi:10.1007/3-540-48660-7_14. Available at https://www.cs.cmu.edu/~fp/papers/cade99.pdf.

[44] Gordon Plotkin & John Power (2003): *Algebraic Operations and Generic Effects*. *Applied Categorical Structures* 11(1), pp. 69–94, doi:10.1023/A:1023064908962. Available at https://homepages.inf.ed.ac.uk/gdp/publications/alg_ops_gen_effects.pdf.

[45] QIR Alliance (2021): *QIR Specification*. Available at https://github.com/qir-alliance/qir-spec. Also see https://qir-alliance.org.

[46] Robert Rand, Kesha Hietala & Michael Hicks (2019): *Formal Verification vs. Quantum Uncertainty*. In: *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, 136, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 12:1–12:11, doi:10.4230/LIPIcs.SNAPL.2019.12.

[47] John C. Reynolds (1975): *A Polymorphic Model of ALGOL*. Unpublished manuscript.

[48] John C. Reynolds (1981): *The Essence of ALGOL*. In: *Proceedings of the International Symposium on Algorithmic Languages*, North-Holland, pp. 345–372. Available at http://www.cs.cmu.edu/afs/cs/user/crary/www/819-f09/Reynolds81.ps. Reprinted as [49].

[49] John C. Reynolds (1997): *The Essence of ALGOL*. In Peter W. O'Hearn & Robert D. Tennent, editors: *ALGOL-like Languages (Volume 1)*, Birkhäuser Boston, pp. 67–88, doi:10.1007/978-1-4612-4118-8_4.

[50] Peter Selinger (2004): *Towards a Quantum Programming Language*. *Mathematical Structures in Computer Science* 14(4), pp. 527–586, doi:10.1017/S0960129504004256. Available at https://www.mathstat.dal.ca/~selinger/papers/papers/qpl.pdf.

[51] Kartik Singhal (2020): *Quantum Hoare Type Theory*. Master's thesis, University of Chicago. arXiv:2012.02154. Homepage: https://ks.cs.uchicago.edu/publication/qhtt-masters/.

[52] Kartik Singhal & John Reppy (2021): *Quantum Hoare Type Theory: Extended Abstract*. In: *Proc. QPL '20*, pp. 291–302, doi:10.4204/EPTCS.340.15. Homepage: https://ks.cs.uchicago.edu/publication/qhtt/.

[53] Frederick Smith, David Walker & Greg Morrisett (2000): *Alias Types*. In: *Proc. ESOP '00*, Springer, pp. 366–381, doi:10.1007/3-540-46425-5_24. Available at https://www.cs.cornell.edu/talc/papers/alias.pdf.

[54] Sam Staton (2013): *An Algebraic Presentation of Predicate Logic (Extended Abstract)*. In: *Foundations of Software Science and Computation Structures (FoSSaCS)*, Springer, pp. 401–417, doi:10.1007/978-3-642-37075-5_26. Available at http://www.cs.ox.ac.uk/people/samuel.staton/papers/fossacs13.pdf.

[55] Sam Staton (2015): *Algebraic Effects, Linearity, and Quantum Programming Languages*. In: *Proc. POPL '15*, ACM, pp. 395–406, doi:10.1145/2676726.2676999. Available at http://www.cs.ox.ac.uk/people/samuel.staton/papers/popl2015.pdf.

[56] Krysta M. Svore, Alfred V. Aho, Andrew W. Cross, Isaac L. Chuang & Igor L. Markov (2006): *A Layered Software Architecture for Quantum Computing Design Tools*. *Computer* 39(1), pp. 74–83, doi:10.1109/MC.2006.4. Available at https://web.eecs.umich.edu/~imarkov/pubs/jour/computer06-q.pdf.

[57] Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher E. Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz & Martin Roetteler (2018): *Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL*. In: *Proceedings of the Real World Domain Specific Languages Workshop 2018*, ACM, pp. 7:1–7:10, doi:10.1145/3183895.3183901. arXiv:1803.00652.

[58] The Univalent Foundations Program (2013): *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

[59] David Walker (2001): *Typed Memory Management*. Ph.D. thesis, Cornell University. Available at https://www.cs.cmu.edu/~dpw/papers/thesis_ps.gz. Order no. 9988176.

[60] David Walker & Greg Morrisett (2001): *Alias Types for Recursive Data Structures*. In: *Types in Compilation*, Springer, pp. 177–206, doi:10.1007/3-540-45332-6_7. Available at https://www.cs.cornell.edu/talc/papers/alias-recursion.pdf.

[61] W. K. Wootters & W. H. Zurek (1982): *A single quantum cannot be cloned*. *Nature* 299(5886), pp. 802–803, doi:10.1038/299802a0. Available at https://copilot.caltech.edu/documents/17036/299802a0.pdf.

[62] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin & Steve Zdancewic (2012): *Formalizing the LLVM Intermediate Representation for Verified Program Transformations*. In: *Proc. POPL '12*, ACM, pp. 427–440, doi:10.1145/2103656.2103709. Available at https://www.cis.upenn.edu/~stevez/papers/ZNMZ12.pdf.

A Derived Forms

In addition to the syntax shown in Figure 2, we use these straightforward derived forms from Harper's language MA [16]:

$$
\begin{aligned}
\{x \leftarrow m_1; m_2\} &\triangleq \texttt{bnd}\, x \leftarrow \texttt{cmd}(m_1)\ ;\ m_2 \\
\{x_1 \leftarrow m_1; \ldots x_{n-1} \leftarrow m_{n-1}; m_n\} &\triangleq \{x_1 \leftarrow m_1; \ldots \{x_{n-1} \leftarrow m_{n-1}; m_n\}\} \\
\{m_1; m_2\} &\triangleq \{\_ \leftarrow m_1; m_2\} \\
\{m_1; \ldots m_{n-1}; m_n\} &\triangleq \{m_1; \ldots \{m_{n-1}; m_n\}\} \\
\texttt{do}\ m &\triangleq \{x \leftarrow m; \texttt{ret}(x)\} \\
\tau_1 \Rightarrow \tau_2 &\triangleq \tau_1 \rightarrow \texttt{cmd}(\tau_2) \\
\texttt{proc}\ (x : \tau)\ \mathrm{m} &\triangleq \lambda(x.\,\texttt{cmd}(m)) \\
\texttt{call}\ e_1(e_2) &\triangleq \texttt{do}\ (\texttt{ap}(e_1; e_2)) \\
\texttt{call}\ e &\triangleq \texttt{call}\ e(\langle\rangle)
\end{aligned}
$$

B Remaining Static and Dynamic Rules

These are standard rules from Harper's PFPL [17] adapted to quantum computation. Note that in the PFPL terminology, we are following the *scoped dynamics* of symbols [17, Ch. 31]. Instead of typed assignables [17, §34.3], we only have a single type of qubit symbols, which we hence do not annotate in the signature, Σ, i.e., the signature only contains active qubit symbols in scope and nothing else. Further, since there are no reference types [17, Ch. 35] except for a single qubit reference type, we do not explicitly state any mobility conditions [17, §31.1]. Under scoped dynamics, qubit references are immobile [17, §35.2]. This mobility restriction is crucial in ensuring the stack discipline for qubit management.

B.1 Type System

We provide the most interesting rules of our type system in Figure 3. We include the following rules here for completeness.

$\boxed{\Gamma \vdash e : \tau}$ *(Expression e has type τ in context Γ)*

$$\textsc{ty-Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{ty-Let}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\,(e_1; x.e_2) : \tau_2}$$

$$\textsc{ty-Lam}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda\,\{\tau_1\}(x.e) : \texttt{fun}\,(\tau_1; \tau_2)} \qquad \textsc{ty-Ap}\ \frac{\Gamma \vdash e_1 : \texttt{fun}\,(\tau_2; \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{ap}\,(e_1; e_2) : \tau}$$

$$\textsc{ty-Pr}\ \frac{\Gamma \vdash e : \texttt{prod}\,(\overline{i \hookrightarrow \tau_i}^{\,i \in 1..n}) \quad 1 \leq i \leq n}{\Gamma \vdash \texttt{proj}\,\langle i \rangle(e) : \tau_i} \qquad \textsc{ty-Tpl}\ \frac{\overline{\Gamma \vdash e_i : \tau_i}^{\,i \in 1..n}}{\Gamma \vdash \texttt{tuple}\,(\overline{i \hookrightarrow e_i}^{\,i \in 1..n}) : \texttt{prod}\,(\overline{i \hookrightarrow \tau_i}^{\,i \in 1..n})}$$

In rule TYS-QLOC and rule VS-QLOC in the next subsection, **qloc**$[q]$ is the value of the reference to an active qubit symbol q. It can be thought of as a classical pointer value indexed by a qubit symbol. The signature plays a role wherever commands or qubits are involved.

$\boxed{\Gamma \vdash_\Sigma e : \tau}$ *(Expression e has type τ relative to the signature)*

$$\textsc{TyS-Cmd}\ \frac{\Gamma \vdash_\Sigma m \doteqdot \tau}{\Gamma \vdash_\Sigma \mathtt{cmd}(m) : \mathtt{cmd}(\tau)} \qquad \textsc{TyS-QLoc}\ \frac{}{\Gamma \vdash_{\Sigma,q} \mathtt{qloc}\langle q\rangle : \mathtt{qref}\langle q\rangle}$$

B.2 Operations Semantics

Pure Classical Sub-language

$\boxed{e\ \mathtt{val}}$ *(e is a value)*

$$\textsc{V-Lam}\ \frac{}{\lambda\{\tau\}(x.e)\,\mathtt{val}} \qquad \textsc{V-Tpl}\ \frac{\overline{e_i\,\mathtt{val}}^{\,i\in 1..n}}{\mathtt{tuple}(\overline{i \hookrightarrow e_i}^{\,i\in 1..n})\,\mathtt{val}}$$

$\boxed{e \longmapsto e'}$ *(e steps to e')*

$$\textsc{Tr-Let}\ \frac{e_1 \longmapsto e_1'}{\mathtt{let}(e_1;x.e_2) \longmapsto \mathtt{let}(e_1';x.e_2)} \qquad \textsc{Tr-LetInstr}\ \frac{e_1\,\mathtt{val}}{\mathtt{let}(e_1;x.e_2) \longmapsto [e_1/x]e_2} \qquad \textsc{Tr-ApL}\ \frac{e_1 \longmapsto e_1'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1';e_2)}$$

$$\textsc{Tr-ApR}\ \frac{e_1\,\mathtt{val} \qquad e_2 \longmapsto e_2'}{\mathtt{ap}(e_1;e_2) \longmapsto \mathtt{ap}(e_1;e_2')} \qquad \textsc{Tr-ApInstr}\ \frac{e_2\,\mathtt{val}}{\mathtt{ap}(\lambda\{\tau_2\}(x.e_1);e_2) \longmapsto [e_2/x]e_1}$$

$$\textsc{Tr-Tpl}\ \frac{\overline{e_i\,\mathtt{val}}^{\,i\in 1..n_a} \qquad e \longmapsto e'}{\mathtt{tuple}(\overline{i \hookrightarrow e_i}^{\,i\in 1..n_a}, k \hookrightarrow e, \overline{j \hookrightarrow e_j}^{\,j\in 1..n_b}) \longmapsto \mathtt{tuple}(\overline{i \hookrightarrow e_i}^{\,i\in 1..n_a}, k \hookrightarrow e', \overline{j \hookrightarrow e_j}^{\,j\in 1..n_b})}$$

$$\textsc{Tr-Pr}\ \frac{e \longmapsto e'}{\mathtt{proj}\langle i\rangle(e) \longmapsto \mathtt{proj}\langle i\rangle(e')} \qquad \textsc{Tr-PrInstr}\ \frac{\mathtt{tuple}(\overline{i \hookrightarrow e_i}^{\,i\in 1..n})\,\mathtt{val} \qquad 1 \le j \le n}{\mathtt{proj}\langle j\rangle(\mathtt{tuple}(\overline{i \hookrightarrow e_i}^{\,i\in 1..n})) \longmapsto e_j}$$

$\boxed{e\ \mathtt{val}_\Sigma}$ *(e is a value relative to Σ)*

$$\textsc{VS-Cmd}\ \frac{}{\mathtt{cmd}(m)\ \mathtt{val}_\Sigma} \qquad \textsc{VS-QLoc}\ \frac{}{\mathtt{qloc}\langle q\rangle\ \mathtt{val}_{\Sigma,q}}$$

Effectful Quantum Sublanguage In the following rules, we do not show the quantum store, preferring the equational dynamics shown in §3.3. In reading these rules, consider the quantum store shape being expanded and restored during the allocation command (as reflected in the signature) and the quantum state being modified during the measurement and the gate application commands.

$\boxed{m\ \mathtt{final}_\Sigma}$ *(Command m is complete)*

$$\textsc{fn-Ret}\ \frac{e\ \mathtt{val}_\Sigma}{\mathtt{ret}\,(e)\ \mathtt{final}_\Sigma}$$

$\boxed{m \longmapsto_\Sigma m'}$ *(Command m steps to m')*

$$\textsc{st-Ret}\ \frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{ret}\,(e) \underset{\Sigma}{\longmapsto} \mathtt{ret}\,(e')} \qquad \textsc{st-Bnd}\ \frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{bnd}\,(e;x.m) \underset{\Sigma}{\longmapsto} \mathtt{bnd}\,(e';x.m)} \qquad \textsc{st-BndInstr}\ \frac{e\ \mathtt{val}_\Sigma}{\mathtt{bnd}\,(\mathtt{cmd}\,(\mathtt{ret}\,(e));x.m) \underset{\Sigma}{\longmapsto} [e/x]m}$$

$$\textsc{st-BndCmd}\ \frac{m_1 \underset{\Sigma}{\longmapsto} m_1'}{\mathtt{bnd}\,(\mathtt{cmd}\,(m_1);x.m_2) \underset{\Sigma}{\longmapsto} \mathtt{bnd}\,(\mathtt{cmd}\,(m_1');x.m_2)} \qquad \textsc{st-NewQRef}\ \frac{m \underset{\Sigma,q}{\longmapsto} m'}{\mathtt{newqref}\,(x.m) \underset{\Sigma}{\longmapsto} \mathtt{newqref}\,(x.m')}$$

$$\textsc{st-NewQRefInstr}\ \frac{e\ \mathtt{val}_\Sigma}{\mathtt{newqref}\,(x.\mathtt{ret}\,(e)) \underset{\Sigma}{\longmapsto} \mathtt{ret}\,(e)} \qquad \textsc{st-GateApRef}\ \frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{gateap}\,\langle U_{2^n}\rangle(e) \underset{\Sigma}{\longmapsto} \mathtt{gateap}\,\langle U_{2^n}\rangle(e')}$$

$$\textsc{st-DiagApRefL}\ \frac{e_1 \underset{\Sigma}{\longmapsto} e_1'}{\mathtt{diagap}\,\langle U_{2^n}, V_{2^n}\rangle(e_1;e_2) \underset{\Sigma}{\longmapsto} \mathtt{diagap}\,\langle U_{2^n}, V_{2^n}\rangle(e_1';e_2)}$$

$$\textsc{st-DiagApRefR}\ \frac{e_1\ \mathtt{val}_\Sigma \qquad e_2 \underset{\Sigma}{\longmapsto} e_2'}{\mathtt{diagap}\,\langle U_{2^n}, V_{2^n}\rangle(e_1;e_2) \underset{\Sigma}{\longmapsto} \mathtt{diagap}\,\langle U_{2^n}, V_{2^n}\rangle(e_1;e_2')} \qquad \textsc{st-MeasRef}\ \frac{e \underset{\Sigma}{\longmapsto} e'}{\mathtt{meas}\,(e) \underset{\Sigma}{\longmapsto} \mathtt{meas}\,(e')}$$

C Correspondence between $\lambda_{Q\#}$ and Staton's quantum language

We can easily translate the quantum-specific fragment of $\lambda_{Q\#}$ to Staton's quantum programming language [55, §5]. Note that the generic effects of his quantum language are equivalent to the algebraic operations (of the algebraic theory) [44] that he uses in his proof of Theorem 11 [55, pp. 12–15, Appendix A]. The translation follows:

$$\begin{aligned} \text{let } q = \underline{\mathsf{new}}() \text{ in } m &\equiv \mathtt{newqref}(q\,.\,m) \\ \underline{\mathsf{measure}}(e) &\equiv \mathtt{meas}(e) \\ \underline{\mathsf{apply}}_U(e) &\equiv \mathtt{gateap}\langle U_{2^n}\rangle(e) \end{aligned}$$

Note that $\texttt{gateap}\langle U_{2^n}\rangle(e)$ subsumes $\texttt{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1\,; e_2)$, just like in Staton's work. In other words, we can define each of Staton's terms using terms of our language.

D An Elaboration Example

Listing 1 shows a sample Q# program. Figure 4 shows the corresponding elaboration to $\lambda_{Q\#}$.

```
namespace Quantum.Kata.Teleportation {

    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);
    }
}
```

Listing 1: Teleportation in Q# (adapted from Quantum Katas [32]).

```
let Entangle be proc (⟨qAlice, qBob⟩ : qref⟨a⟩ × qref⟨b⟩) {
    apply_H(qAlice);
    apply_{D(I_2,X)}(qAlice ; qBob)} in
let SendMsg be proc (⟨qAlice, qMsg⟩ : qref⟨a⟩ × qref⟨m⟩) {
    apply_{D(I_2,X)}(qMsg ; qAlice);
    apply_H(qMsg);
    ret(⟨cmd(meas(qMsg)), cmd(meas(qAlice))⟩)} in
let DecodeMsg be proc (⟨qBob, ⟨b_1, b_2⟩⟩ : qref⟨b⟩ × (bool × bool)) {
    if b_1 then apply_Z(qBob) else ⟨⟩;
    if b_2 then apply_X(qBob) else ⟨⟩} in
let Teleport be proc (⟨qAlice, qBob, qMsg⟩ : qref⟨a⟩ × qref⟨b⟩ × qref⟨m⟩) {
    call Entangle(⟨qAlice, qBob⟩);
    classicalBits ← call SendMsg(⟨qAlice, qMsg⟩);
    call DecodeMsg(⟨qBob, classicalBits⟩)} in ⟨⟩
```

Figure 4: $\lambda_{Q\#}$ elaboration of the Q# program in Listing 1.