Q# as a Quantum Algorithmic Language

Kartik Singhal ⚡️✍️ ⚡️
University of Chicago, USA

Kesha Hietala ⚡️✍️ ⚡️
University of Maryland, College Park, USA

Sarah Marshall ⚡️✍️ ⚡️
Microsoft Quantum, Redmond, USA

Robert Rand ⚡️✍️ ⚡️
University of Chicago, USA

Abstract

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 complete algebraic theory by Staton.

2012 ACM Subject Classification Theory of computation → Quantum computation theory; Theory of computation → Type structures; Software and its engineering → Syntax; Software and its engineering → Semantics; Theory of computation → Algebraic semantics

Keywords and phrases Q#, quantum computation, type systems, formal specification, formal language definitions, semantics and reasoning, quantum programming languages, language design

Supplementary Material Software (Complete grammar and ongoing mechanization): https://github.com/k4rtik/lambda-qs

Funding This work is funded in part by EPiQC, an NSF Expedition in Computing, under Award Number CCF-1730449; in part by the US Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040; and in part by the Air Force Office of Scientific Research under Grant Number FA95502110051.

Acknowledgements We thank Bettina Heim and Alan Geller for helping us get this project off the ground, Bob Harper for several fruitful conversations, and Jennifer Paykin and Sam Staton for critical feedback on early drafts. We also benefited from discussions with Amy Felty, Michael Hicks, Ralf Jung, Ohad Kammar, Alex Kavvos, Brigitte Pientka, and Stephanie Weirich.

1 Introduction

Microsoft’s Q# programming language [61] is one of the most sophisticated quantum programming languages that has 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 with many growing programming languages—maintaining correctness, ease of use, and intuitive understanding while evolving to meet users’ needs.

Quantum programming languages, in particular, face unique challenges that are not present in classical languages. Quantum algorithms are generally more challenging to design and reason about than classical ones since they use quantum phenomena like superposition...
Q# as a Quantum Algorithmic Language

and entanglement. They are also difficult to test and debug. Simulation of quantum programs on classical computers is slow and limited to a handful of qubits. At the same time, 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, the 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: exposing its treatment of qubits as references, its stack-like qubit management that enables reasoning about the quantum state in a local manner, and safe synthesis of effectful quantum and pure classical computation. The latter two are inherent to many ALGOL-like languages [49, 50].

Contributions

In 2018, when introducing Q#, its designers stated that as opposed to several existing circuit definition languages, “Q# is an algorithm definition language” [61]; 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) [18]. 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 complete equational theory of quantum computation by Staton [59].
- 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); we introduce \( \lambda_{Q#} \) along with its syntax and semantics (§3); we describe how \( \lambda_{Q#} \) is faithful the surface Q# language (§4); and we conclude with a discussion of 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# [61], a modern, self-contained quantum programming language that
teaches quantum computing through its *quantum katas* [32] and boasts a large community of developers. The second is Sam Staton’s equational theory for quantum programs [59], which builds upon a large body of work on algebraic effects and semantics of quantum programming languages.

### 2.1 The Q# Programming Language

Q# [61] 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.

Q# contains a blend of functional and imperative features. For classical data, Q# follows the so-called value semantics [22] which is better known as referential transparency in the PL community. 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 [12, 13]—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 \texttt{Unit}, after altering the quantum state.

Callable 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 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 (\texttt{Adjoint U}) and controlled (\texttt{Controlled U}) versions of some unitary operation, \texttt{U}. The functors (in Q#’s terminology\footnote{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.’}) \texttt{Adjoint} and \texttt{Controlled} are parametric in their arguments. Still, functors only accept those operations that have \texttt{Adj} or \texttt{Ctl} characteristics associated with them.

Q# follows the QRAM model of computation [27] which assumes an infinite supply of logical qubits from which the programmer can obtain a reference to a new qubit by calling the \texttt{use} command. Qubits are hence allocated and deallocated in a stack-like manner, where the lifetime of a unique qubit is equivalent to the lexical scope of the \texttt{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. Listing 2 shows a minimal example that passes the type checker but fails at runtime (in a simulator).

Programmers are allowed to create new bindings using \texttt{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 [64], which forbids duplication of qubits. In Listing 3, both \texttt{q1} and \texttt{q2} refer to the same qubit. Applying \texttt{CNOT} with \texttt{q1} as the control and \texttt{q2} as the target is equivalent to cloning the underlying qubit. Currently, Q# can not prevent this issue statically.

An informal specification of the Q# language was recently published [22, 23]. 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 [59] presents a substructural (linear) version of his framework for “parameterized algebraic theories” [58]. 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 that applies to the QRAM model [27], 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 [55, 56], and, to our advantage, Q# [61].

Here we reproduce Staton’s theory of a “quantum local store” for reference; we will see in §3.3 how this algebraic theory helps us describe the equational dynamics of $\lambda_Q$.

Generic Effects

Staton adds the following generic effects to a standard linear type theory and obtains a language similar to Selinger’s QPL [53].

- $\Gamma \vdash \text{new}() : \text{qubit}$
- $\Gamma \vdash t : \text{qubit} \otimes n$
- $\Gamma \vdash \text{apply}_U(t) : \text{qubit} \otimes n$
- $\Gamma \vdash \text{measure}(t) : \text{bool}$

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 [59, §6.2].

Axioms relating unitary gates and measurement:

- (A) $\text{measure}(\text{apply}_X(a)) \equiv \neg \text{measure}(a)$
- (B) $\text{let } (a', x') \text{ be } \text{apply}_{D(U,V)}(a, x) \text{ in } (\text{measure}(a'), x') \equiv$
  
  \[
  \begin{cases} 
  (0, \text{apply}_{U}(x)) \text{ if } \text{measure}(a) = 0 \\
  (1, \text{apply}_{V}(x)) \text{ else } \end{cases}
  \]
- (C) $\text{discard}(\text{apply}_U(x)) \equiv \text{discard}(x)$

Axioms relating allocation with unitaries and measurement:

- (D) $\text{measure}(\text{new}()) \equiv 0$
- (E) $\text{apply}_{D(U,V)}(\text{new}(), x) \equiv (\text{new}(), \text{apply}_U(x))$

where $D(U,V) = U \oplus V = \left( \begin{array}{c|c} U & 0 \\ \hline 0 & V \end{array} \right)$ is a block diagonal matrix (equivalently, the direct sum).

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, then applying a unitary is the same as doing nothing. Axiom (D) states that

\footnote{However, the stack-like management of qubits is unique to Q#.}
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 [21] developed a well-typed internal language for Standard ML, defined an elaboration relation between the external language and this internal language, and proved properties of the metatheory of the language using the internal language. Harper and collaborators [29, 6] followed this work with the mechanization of the metatheory using the Twelf logical framework [42]. 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-level 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) [18], 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 (IA) [49, 50], which is the basis for the quantum language IQu [38], 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. We will 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. For simplicity, we do not consider borrowing in this work as it is an optimization concern that lets a programmer use dirty ancillae in their code. The only allowed operations on qubits are gate application and measurement.

#### 3.1 Syntax

Figure 1 presents the syntax of $\lambda_Q#$. We divide the grammar into a monadic and effectful command language and a pure expression language, the familiar simply-typed lambda calculus extended with encapsulated commands. We precisely specify the binding structure of the syntax following the notion of abstract binding trees [19, Chapter 1]. We show the convenience syntax on the right in blue color along with a description of productions where
\[\tau ::= \begin{align*}
qref \,[\,q\,] & \quad \text{qref } q, \text{ qubit reference type} \\
\text{arr} \,(\tau_1;\tau_2) & \quad \tau_1 \to \tau_2 \\
\text{cmd} \,(\tau) & \quad \tau \text{ cmd} \\
\text{prod} \,(\underbrace{\tau_i \hookrightarrow \tau_i}_{i \in \{1..n\}}) & \quad \times_{i \in L} \tau_i \\
\text{bool} & \quad \text{bool} \\
\text{unit} & \quad \text{unit}
\end{align*}\]

\[\epsilon ::= \begin{align*}
x & \quad x \\
\text{let} \,(e_1;\underbrace{x.e}_{2}) & \quad \text{let } x \text{ be } e_1 \text{ in } e_2 \\
\text{lam} \,\{\tau\} \,(x.e) & \quad \lambda x : \tau \text{ e} \\
\text{ap} \,(e_1;e_2) & \quad e_1(e_2) \\
\text{cmd} \,(m) & \quad \text{cmd } m, \text{ encapsulation} \\
\text{tpl} \,(\underbrace{l_i \hookrightarrow e_i}_{i \in \{1..n\}}) & \quad \langle e_i \rangle_{i \in L}, \text{ tuple} \\
\text{pr} \,[l](e) & \quad e \cdot l, \text{ projection} \\
\text{true} & \quad \text{true} \\
\text{false} & \quad \text{false} \\
\text{if} \,(e;e_1;e_2) & \quad \text{if } e \text{ then } e_1 \text{ else } e_2 \\
\text{not} \,e & \quad \neg \text{ e} \\
\text{triv} & \quad \langle \rangle
\end{align*}\]

\[m ::= \begin{align*}
\text{ret} \,(e) & \quad \text{ret } e, \text{ return} \\
\text{bnd} \,(e;x.m) & \quad \text{bnd } x \leftarrow e; m, \text{ sequencing} \\
\text{newqref} \,(x.m) & \quad \text{new } x \text{ in } m, \text{ new qubit reference} \\
\text{gateap} \,[U_{2^n} \, e] & \quad \text{U } (e), \text{ gate application} \\
\text{diagap} \,[U_{2^n},V_{2^n}] \,(e_1;e_2) & \quad D(U,V)(e_1, e_2), \text{ block diagonal} \\
\text{meas} \,(e) & \quad \text{meas}(e), \text{ measurement}
\end{align*}\]

\begin{itemize}
\item Figure 1 \(\lambda_{Q\#}\) grammar showing types, pure expressions, and effectful commands. Some operators are indexed by symbolic parameters, which are marked by square brackets.
\end{itemize}

unclear. We also include some standard derived forms from Harper’s language \(\text{MA}\), which are shown in Appendix A. We will use the convenience syntax along with the derived forms in our presentation wherever there is no possibility of confusion.

We do not consider the pure expression language in detail, excluding all classical base types except \texttt{unit} and \texttt{bool}. This way, we can focus on the interesting quantum-classical interface at play in Q#. The qubit reference type \texttt{qref}[q] is a singleton type (inspired by their use in Alias Types [57]). Qubit symbols are shown in orange color 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 italics) are parametric to the grammar (similar to Q#, which does not have a preference for a specific gate set) and are typed as \(U : \times^n \texttt{qref } q_i \rightarrow \texttt{unit cmd}\), where the arity of the product is \(n\) and \(\text{dim}(U) = 2^n\). The \(U(e)\) 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
Figure 2 Typing of commands. \( \Gamma \) is the standard typing context, and the signature, \( \Sigma \), keeps track of qubit symbols in scope. Note that each qubit symbol is unique. See other rules in Appendix B.1.

using block diagonals, e.g., \( \text{CNOT} \equiv \text{D}(I_2, X) \) (in circuit notation) and are typed as \( \text{D}(U, V) : \times^{n+1} \text{qref} \ q_i \rightarrow \text{unit cmd} \), where \( \dim(U) = \dim(V) = 2^n \). It is understood that 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 \( \lambda_{\text{Q#}} \) is the usual simply-typed lambda calculus, so we will not say much about it here. We show typing rules for the effectful portion of \( \lambda_{\text{Q#}} \) in Figure 2.

All of our command typing judgments are parameterized by a signature, \( \Sigma \), 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, 49, 50]. 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 the local state. Another view is to think of the commands as being parametrically polymorphic [33, 34, 35] to the store, an idea considered by Reynolds as early as 1975 [5, 48], even before store shapes. Still, we prefer the signature-based approach to keep our language closer to Harper’s MA.

These are the two standard rules for monadic return and bind operations and four rules specific to quantum computation.

Rule cmd-NewQRef shows how the signature is extended with a new qubit symbol on the execution of newqref, which allocates a new logical qubit \( q \) and immediately returns a reference to it. Its binding structure ensures that the lifetime of the newly allocated qubit is

\(^3\) Store shapes follow laws similar to what are known as lenses in the current literature [10].
equal to its lexical scope. This choice of allocation form ensures a strict stack discipline and provides safe and automatic management of qubits.

Rules \textsc{cmd-GateApRef} and \textsc{cmd-DiagApRef} show how each of these rules enforces the constraint that the input qubit references are distinct. Rule \textsc{cmd-MeasRef} shows that we only support the measurement of a single qubit at a time.

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 additional safety properties currently not offered by Q\#'s type system:

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

It is clear from rules \textsc{cmd-GateApRef} and \textsc{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 2.** The unsafe code fragment from Listing 3 can be written in $\lambda_{Q\#}$ syntax as:

\[
\text{newqref}(q_1, \text{ret(let}(q_1; q_2. \text{cmd(diagap}[I_2, X](q_1, q_2))))
\]

or with some syntactic sugar as:

\[
\text{new } q_1 \text{ in ret let } q_2 \text{ be } q_1 \text{ in cmd } D(I_2, X)(q_1, q_2)
\]

Since the type of the qubit reference in $\lambda_{Q\#}$, $\text{qref}[q]$, 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 will allow it to pass.

▶ **Proposition 3.** $\lambda_{Q\#}$ statically ensures safe memory management and disallows dangling qubit references.

The allocation command, 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 \textsc{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 4.** Using $\lambda_{Q\#}$ convenience syntax and the derived forms from Appendix A, the unsafe code fragment shown in Listing 2 can be written as:

\[
\text{let NewQubit be proc () { new x in ret x } in ()}
\]

Here, while $\Gamma, x : \text{qref } q \vdash \Sigma, q \text{ ret } x \sim \text{qref } q$, meaning that the premise of rule \textsc{cmd-NewQRef} holds with the extended context, $\Gamma \not\vdash_{\Sigma} \text{ new } x \text{ in ret } x \sim \text{qref } q$, since the signature $\Sigma$ no longer contains the qubit $q$. Hence, this invalid program is also ruled out.
3.3 Dynamic Semantics

For the dynamics of the effectful quantum fragment, we rely on Staton’s equational theory for quantum local store [59, §6.2]. Unlike Staton’s language, our unitary operation does not return the qubit but modifies it in place. In this presentation, we use the convenience syntax from Figure 1 and 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

\[
a : \text{qref } q \vdash \{X(a); \text{meas}(a)\} \equiv \neg \text{do }\{\text{meas}(a)\} \quad (A)
\]
\[
a : \text{qref } q, b : \bigotimes^n \text{qref } r_i \vdash \{D(U,V)(a,b); \text{meas}(a); \text{ret }\} \equiv
\]
\[
\text{do }\{x \leftarrow \text{meas}(a); \text{ret if } x \text{ then } \text{cmd } V(b) \text{ else } \text{cmd } U(b)\} \quad (B)
\]
\[
\cdot \vdash \text{do }\{\text{new } a \text{ in } \text{meas}(a)\} \equiv \text{false} \quad (D)
\]
\[
b : \text{qref } q \vdash \{\text{new } a \text{ in } D(U,V)(a,b)\} \equiv \text{do }\{U(b); \text{new } a \text{ in } \text{ret }\} \quad (E)
\]

We mentioned in §2.2 that one could omit Staton’s axiom (C) for the QRAM scenario. This is because, in the QRAM model, the discard operation just forgets the name of the qubit. In Q# and λQ♯, we may consider an equivalent operation: qubit references are automatically forgotten when they reach the end of their lexical scope. A variant of axiom (C) holds for both Staton’s theory for a quantum local store (and our λQ♯) that says one can ignore the global phase.

Administrative Axioms

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

\[
m_1 : \tau_1 \text{ cmd } m_2 : \tau_2 \text{ cmd } \vdash \text{do }\{\text{new } a \text{ in } \text{new } b \text{ in } m_1; \text{SWAP}(a,b); m_2\} \equiv
\]
\[
\text{do }\{\text{new } a \text{ in } \text{new } b \text{ in } m_1; \text{let } (b,a) \text{ be } (a,b) \text{ in } \text{cmd } m_2\} \quad (F)
\]
\[
e : \bigotimes^n \text{qref } q_i \vdash \{I_{\otimes^n}(e)\} \equiv \langle \rangle \quad (G)
\]
\[
e : \bigotimes^n \text{qref } q_i \vdash \{VU(e)\} \equiv \{U(e); V(e)\} \quad (H)
\]
\[
e_1 : \bigotimes^m \text{qref } q_i, e_2 : \bigotimes^n \text{qref } r_i \vdash \{U_{\otimes(1\ldots n)}(e_1,e_2)\} \equiv \{U(e_1); V(e_2)\} \quad (I)
\]

Selinger [53] notes that the SWAP gate is equivalent to classically renaming the qubit references, which captures the intuition behind equation (F), although, in our case, we have to ensure that the scope of the 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 product (horizontal and vertical composition, respectively, in circuit notation).

Like Staton, we also state the commutativity equations that hold for λQ♯:

\[
a : \text{qref } q, b : \text{qref } r, m : \tau \text{ cmd } \vdash \{x \leftarrow \text{meas}(a); y \leftarrow \text{meas}(b); m\} \equiv
\]
\[
\text{do }\{y \leftarrow \text{meas}(b); x \leftarrow \text{meas}(a); m\} \quad (J)
\]
\[
m : \tau \text{ cmd } \vdash \text{do }\{\text{new } a \text{ in } \text{new } b \text{ in } m\} \equiv \text{do }\{\text{new } b \text{ in } \text{new } a \text{ in } m\} \quad (K)
\]
\[
b : \text{qref } q, m : \tau \text{ cmd } \vdash \{\text{new } a \text{ in } y \leftarrow \text{meas}(b); m\} \equiv
\]
\[
\text{do }\{y \leftarrow \text{meas}(b); \text{new } a \text{ in } m\} \quad (L)
\]
Now that we have shown that the quantum portion of $\lambda Q$ is equivalent to Staton’s quantum programming language [59, §5] and corresponding program equations with his theory of quantum local store, we can state a variant of Staton’s result [59, Theorem 11]:

▶ Theorem 5 (Universality of $\lambda Q$). For any linear map $f : M_{2^n_1} \oplus \cdots \oplus M_{2^n_k} \rightarrow M_{2^p}$ 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 [59], which is a variation on Selinger’s Theorem 6.4 [53]. The proof relies on the correspondence between Staton’s simple quantum language and a fragment of $\lambda Q$, where the translation is straightforward.

▶ Theorem 6 (Completeness). Assuming an axiomatization of unitaries, if two terms $t$ and $u$ have equivalent interpretation in a common context, $\Gamma$, then $\Gamma \vdash t \equiv u$ is derivable.

Note that since Q# is parameterized over gate sets, we do 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 lambda calculus, which are required because our axioms do not live in isolation but are written as typed expressions in 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 convenience syntax from Figure 1 and the derived forms from Appendix A. The elaboration of the Q# teleport example from Listing 1 is shown in Figure 3.

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 qref, we must 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 3, $a$, $b$, 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 characteristics and inlines the corresponding operation specialization, converting it to a unitary operator. We need to do this during elaboration because we do not encode characteristic information or specializations directly in $\lambda Q$. An adjointable Q# operation with type $\langle\text{Qubit}, \ldots, \text{Qubit}\rangle \Rightarrow \text{Unit}$ 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) means that 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 {} 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.

In our initial attempt to get the foundations right and for the sake of simplicity, we do not yet support several Q# features: namespaces; operation characteristics; custom operation
Table 1 Select Q# to λQ# 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. [ ] is the elaboration function and mat(·) converts a λQ# expression to its corresponding unitary operator. In the rule for Qubit, q is determined from the elaboration context.

<table>
<thead>
<tr>
<th>Q# Syntax</th>
<th>λQ# Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(τ₁,...,τₙ)]</td>
<td>×ⁿ [τ₁]</td>
</tr>
<tr>
<td>τ₁ → τ₂</td>
<td>[τ₁] → [τ₂]</td>
</tr>
<tr>
<td>τ₁ ≫ τ₂</td>
<td>[τ₁] ⇒ [τ₂]</td>
</tr>
<tr>
<td>[Bool] and [Result]</td>
<td>bool</td>
</tr>
<tr>
<td>[Qubit]</td>
<td>qref q</td>
</tr>
<tr>
<td>[Unit]</td>
<td>unit</td>
</tr>
<tr>
<td>[function f (x₁: τ₁,...) : τ { e } ...]</td>
<td>let f be λ((x₁): ×ⁿ [τ₁]) e in [ ... ]</td>
</tr>
<tr>
<td>[operation f (x₁: τ₁,...) : τ { s } ...]</td>
<td>let f be proc((x₁): ×ⁿ [τ₁]) s in [ ... ]</td>
</tr>
<tr>
<td>[return e]</td>
<td>ret [e]</td>
</tr>
<tr>
<td>[let x = e; ...]</td>
<td>let x be [e] in [ ... ]</td>
</tr>
<tr>
<td>[if e { s₁ } else { s₂ }]</td>
<td>if [e] then cmd [s₁] else cmd [s₂]</td>
</tr>
<tr>
<td>[use q = Qubit { s }]</td>
<td>new q in [s]</td>
</tr>
<tr>
<td>[[Adjoint] e₁ (e₂)]</td>
<td>Uᵀ([e₂]), where U = mat([e₁])</td>
</tr>
<tr>
<td>[[Controlled e₁ (q, e₂)] [e₁] (e₂)]</td>
<td>D(I₂ⁿ, U)(q, [e₂]), where U = mat([e₁])</td>
</tr>
<tr>
<td>[(e₁,...,eₙ)]</td>
<td>[e₁],...,[eₙ]</td>
</tr>
<tr>
<td>[true] and [One]</td>
<td>true</td>
</tr>
<tr>
<td>[false] and [Zero]</td>
<td>false</td>
</tr>
</tbody>
</table>

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 Bool, Result, and Unit; fail; mutable bindings; iteration using for, while, or repeat; and within-apply blocks (which apply an operation and its adjoint).

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 [21, 29], Java [24], JavaScript [17], Rust [26, 25], and, most recently, Go [16]. As we mentioned in §3, we more or less followed the pioneering methodology of the formalization and mechanization of the definition of Standard ML [31] by identifying a well-founded internal language and performing all metatheoretical reasoning on that core. All of these projects demonstrate the extent to which it is possible to distill large and complex languages into formal and faithful cores. Featherweight Java and Go serve as examples of industry-scale languages in mass use benefitting from formalization and academic study in the form of extensions such as generics (polymorphism) first being investigated on smaller cores of the respective languages before being adopted in production over the years. We hope our work serves as a similar playground for extensions. 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. The innovation of this work was to invent a new methodology called tested semantics to compare the output of real-world code with that of the equivalent code in its formal core. We say more in the next section as we plan to adopt
let Entangle be proc ((qAlice, qBob) : qref a × qref b) {
   \( H(qAlice); \)
   \( D(I_2, X)(qAlice, qBob) \)
} in

let SendMsg be proc ((qAlice, qMsg) : qref a × qref m) {
   \( D(I_2, X)(qMsg, qAlice); \)
   \( H(qMsg); \)
   ret (cmd meas(qMsg), cmd meas(qAlice))\}

let DecodeMsg be proc ((qBob, (b_1, b_2)) : qref b × (bool × bool)) {
   if b_1 then \( Z(qBob) \) else \( \langle \rangle \); \)
   if b_2 then \( X(qBob) \) else \( \langle \rangle \)\}

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))\}

\[ \text{Figure 3} \] \( \lambda_{Q\#} \) elaboration of the Q\# program in Listing 1.

some of these ideas in our work. The RustBelt project focused on mechanically verifying the safety claims of the Rust programming language and showed that even a language with a pretty sophisticated type system could be studied at scale and formal assurance provided even for its libraries containing low-level unsafe code.

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 [40] build upon Staton’s work and present an equational theory for quantum computation embedded inside homotopy type theory (HoTT). The essential idea is to treat unitaries as higher inductive paths that 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. [41] 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 [28]. 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 work (§6).
Formal Quantum Programming Languages

Research into the formal semantics of quantum programming languages dates back to Selinger’s QPL [53], a quantum programming language with both denotational semantics in terms of partial density matrices (in which traces do not necessarily sum to one, corresponding to non-termination) and a categorical model as a symmetric monoidal category with traced finite coproducts. Selinger and Valiron’s subsequent work on quantum lambda calculus [54] pioneered the use of affine types. This substructural type system prohibits copying to enforce no cloning by treating qubits as single-use resources. Paykin et al.’s QWire [39] mixed linear and dependent types to guarantee that all qubits are used exactly once, and that quantum circuits compose properly. QWire was initially endowed with denotational semantics in terms of density matrices but later given a categorical semantics by Rennela and Staton [47] as an enriched category, in which QWire circuits are objects within the category of the circuit metaprogramming language. Finally, a series of works on “Proto-Quippers” [52, 51, 30, 11] study the semantics and type systems of quantum languages as fragments of the Quipper language [15].

A key similarity between the many successors to QPL (which itself is a flowchart language) is that they are fundamentally circuit building languages. That is, they generate quantum circuits for execution on a quantum device, following Knill’s QRAM model [27]. Since qubits are objects in these languages, they are necessarily one-time-use objects. By contrast, Q# and λQ# have only references to qubits and hence face a distinct challenge of tracking those references. And while these languages feature a broad range of semantic interpretations, none of them have an equational theory.

A final QPL descendant worth addressing here is IQu [38], which extends Idealized Algol with quantum circuits and quantum variables, much as we extend Harper’s Modernized Algol. Like λQ#, 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 quantum states for their manipulation.

6 Conclusion and Perspectives

We presented our work on defining a core calculus for the Q# programming language, dubbed λQ#. We maintained a separation between the quantum effectful and the pure expression sub-languages, exposed the monadic nature of computation inherent in Q#, made qubit aliasing and block structure explicit, and presented an equational semantics for λQ# building upon Staton’s algebraic theory for quantum local store [59].

A formal specification of the whole Q# language still requires more work. Some extensions are straightforward; for example, 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 other types. Other features are more challenging, including arrays, slices, iteration, polymorphism, and patterns like within-apply and repeat-until-success. 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 we may not be able to distinguish statically); we may need to introduce the notion of a phase
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 [9], 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. Other domain-specific metatheory tools and libraries, including Twelf [42], Hybrid [8], and Beluga [43], also offer no such support. More general frameworks like the Coq proof assistant require the burdensome handling of variable assumptions. However, we still have hope with tools like LNgen [3, 4] that automate the generation of hundreds of lemmas.

Another way we can make our formalization more rigorous is by taking the tested-semantics approach pioneered in the \( \lambda_{JS} \) project [17], where the outputs generated by programs written in the surface grammar are compared against those produced by the core after translation. With access to a massive corpus of open-source Q# code in the form of Microsoft’s quantum libraries and katas, we are optimistic that this approach will yield results. We have started taking small steps in this direction along with collaborators.\(^4\)

A major goal of this project is to form a playground for prototyping extensions to the type system of Q#. To illustrate one problem, a peculiar decision in Q# is to allow uncontrolled aliasing of qubits in support of user-friendly features such as arrays of qubits. While convenient, reasoning about interference freedom for arrays can be notoriously hard; specifically, our solution to enforce no cloning inspired by Alias Types [57, 63] does not scale to arrays [62, §3.5.1]. We are extending our \( \lambda_{Q#} \) type checker with a constraint solver to evaluate potential solutions for common scenarios that occur in practice in Q# library code. Depending on the complexity of the array indexing used in practice, we may use a concrete, natural number inequality checker, a simple symbolic numerical solver, or a full-fledged SMT solver like Z3 [7] to guarantee qubit distinctness.

One of our insights from this project is that even though quantum computation is a fundamentally a new abstraction, many classical techniques both from programming languages and compilers communities can be adapted to the quantum setting [60]. Q# and its Quantum Development Kit (QDK) are a significant example of the realization of 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 [14], 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 theVerified LLVM (Vellvm) project [65]. 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 towards our broader vision of a fully verified quantum stack [46].

---

References


\(^4\) See ongoing implementations of elaboration and a type checker for \( \lambda_{Q#} \) at https://github.com/k4rtik/lambda-qs/tree/main/src/elab.
Q# as a Quantum Algorithmic Language


### A Derived Forms

These are straightforward derived forms from Harper’s MA [18]:

\[
\begin{align*}
\{ x \leftarrow m_1; m_2 \} & \triangleq \text{bnd } x \leftarrow \text{cmd } m_1; m_2 \\
\{ x_1 \leftarrow m_1; \ldots; x_{n-1} \leftarrow m_{n-1}; m_n \} & \triangleq \{ x_1 \leftarrow m_1; \ldots; \{ x_{n-1} \leftarrow m_{n-1}; m_n \} \} \\
\{ m_1; m_2 \} & \triangleq \{ m_1; m_2 \} \\
\{ m_1; \ldots; m_n \} & \triangleq \{ m_1; \ldots; m_n \} \\
do m & \triangleq \{ x \leftarrow m; \text{ret } x \} \\
\tau_1 \Rightarrow \tau_2 & \triangleq \tau_1 \rightarrow \text{cmd } \tau_2 \\
\text{proc } (x : \tau) m & \triangleq \lambda (x : \tau) \text{ cmd } m \\
call e_1(e_2) & \triangleq \text{do } (e_1(e_2)) \\
call e & \triangleq \text{call } e(\langle \rangle)
\end{align*}
\]

### B Remaining Static and Dynamic Rules

These are standard rules from Harper’s PFPL [19] adapted to quantum computation. Note that in the PFPL parlance, we are following the scoped dynamics of symbols [19, Ch. 31]. Instead of typed assignables [19, §34.3], we only have a single type of qubit symbols, which we hence do not annotate in the signature, \( \Sigma \), i.e., the signature only contains active qubit symbols in scope and nothing else. Further, since there are no reference types [19, Ch. 35] except for a single qubit reference type, we do not explicitly state any mobility conditions [19, §31.1]. Under scoped dynamics, qubit references are immobile [19, §35.2]. This mobility restriction is crucial to ensure the stack discipline for qubit management.

#### B.1 Statics

\[
\begin{align*}
\Gamma & \vdash e : \tau \\
\text{TY-VAR} & \\
\Gamma, x : \tau & \vdash x : \tau \\
\text{TY-LET} & \\
\Gamma & \vdash e_1 : \tau_1 \\
\Gamma, x : \tau & \vdash e_1 \vdash e_2 : \tau_2 \\
\Gamma & \vdash \text{let } (e_1; e_2) : \tau_2 \\
\text{TY-LAM} & \\
\Gamma & \vdash \tau_1 \vdash e : \tau_2 \\
\Gamma & \vdash \text{lam } \{ \tau_1 \} (x.e) : \text{arr } (\tau_1; \tau_2) \\
\text{TY-AP} & \\
\Gamma & \vdash e_1 : \text{arr } (\tau_2; \tau) \\
\Gamma & \vdash e_2 : \tau_2 \\
\Gamma & \vdash \text{ap } (e_1; e_2) : \tau \\
\text{TY-PR} & \\
\Gamma & \vdash e : \text{prod } (\tau_i \cdots \tau_i) \\
1 \leq i \leq n \\
\Gamma & \vdash \text{pr } [l_1][e] : \tau_i \\
\text{TY-TPL} & \\
\Gamma & \vdash e_i : \tau_i \\
1 \leq i \leq n \\
\Gamma & \vdash \text{tpl } (\tau_i \cdots \tau_i) : \text{prod } (\tau_i \cdots \tau_i)
\end{align*}
\]

In rule \( \text{TYS-QLOC} \) and rule \( \text{VS-QLOC} \) in the next subsection, \( \text{qloc}[q] \) 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.

\[
\Gamma \vdash \Sigma \ e : \tau \quad \text{(Expression } e \text{ has type } \tau \text{ relative to the signature)}
\]

\[
\text{TYS-Cmd} \\
\Gamma \vdash \Sigma \ m \sim \tau \\
\Gamma \vdash \Sigma \ \text{cmd} (m) : \text{cmd} (\tau)
\]

\[
\text{TYS-QLoc} \\
\Gamma \vdash \Sigma, q \ \text{qloc} [q] : \text{qref} [q]
\]

### B.2 Dynamics

#### Pure Classical Sub-language

\[
e \text{val} \\
\text{V-Lam} \\
\text{lam} \{\tau\}(x.e) \text{val}
\]

\[
e \mapsto e' \\
\text{TR-LEt} \\
\text{let} \ (e_1; x.e_2) \mapsto \text{let} \ (e_1' ; x.e_2)
\]

\[
\text{TR-APR} \\
\text{ap} \ (e_1; e_2) \mapsto \text{ap} \ (e_1'; e_2)
\]

\[
\text{TR-APInstr} \\
\text{ap} \ (\text{lam} \ \{\tau\} (x.e_1); e_2) \mapsto [e_2/x]e_1
\]

\[
\text{TR-Tpl} \\
\text{tpl} \ (l_1 \mapsto e_i \in \{1..n\}, l \mapsto e_j) \mapsto \text{tpl} \ (l_1 \mapsto e_i^{(1..n)}, l \mapsto e_j', l_j' \mapsto e_j')
\]

\[
\text{TR-PR} \\
\text{pr} [l_i] (e) \mapsto \text{pr} [l_i] (e')
\]

\[
\text{TR-PRInstr} \\
\text{pr} [l_j] (\text{tpl} (l_1 \mapsto e_i^{(1..n)})) \mapsto e_j
\]

\[
e \text{val} \Sigma \\
\text{vS-Cmd} \\
\text{cmd} (m) \text{val}_\Sigma
\]

\[
\text{vS-QLoc} \\
\text{qloc} [q] \text{val}_{\Sigma,q}
\]
Effectful Quantum Sub-language

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.

\[
\begin{align*}
\text{st-Ret} & \quad e @ \Sigma \\
\text{ret} (e) & \quad \Sigma \rightarrow \Sigma \\
\end{align*}
\]

\[
\begin{align*}
\text{st-Bnd} & \quad e @ \Sigma \\
\text{bnd} (e; x.m) & \quad \Sigma \rightarrow \Sigma \\
\end{align*}
\]

\[
\begin{align*}
\text{st-BndInstr} & \quad e @ \Sigma \\
\text{bnd} (\text{cmd} (\text{ret} (e)); x.m) & \quad \Sigma \rightarrow [e/x]m
\end{align*}
\]

\[
\begin{align*}
\text{st-BndCmd} & \quad m_1 @ \Sigma \\
\text{bnd} (\text{cmd} (m_1); x.m_2) & \quad \Sigma \rightarrow \text{bnd} (\text{cmd} (m'_1); x.m_2)
\end{align*}
\]

\[
\begin{align*}
\text{st-NewQRef} & \quad m @ \Sigma \\
\text{newqref} (x.m) & \quad \Sigma \rightarrow \text{newqref} (x.m')
\end{align*}
\]

\[
\begin{align*}
\text{st-NewQRefInstr} & \quad e @ \Sigma \\
\text{newqref} (x.\text{ret} (e)) & \quad \Sigma \rightarrow \text{ret} (e)
\end{align*}
\]

\[
\begin{align*}
\text{st-GateApRef} & \quad e @ \Sigma \\
ge\text{gateap} [U_2^n] (e) & \quad \Sigma \rightarrow \text{gateap} [U_2^n] (e')
\end{align*}
\]

\[
\begin{align*}
\text{st-DiagApRefL} & \quad e_1 @ \Sigma \\
\text{diagap} [U_2^n, V_2^n] (e_1; e_2) & \quad \Sigma \rightarrow \text{diagap} [U_2^n, V_2^n] (e'_1; e'_2)
\end{align*}
\]

\[
\begin{align*}
\text{st-DiagApRefR} & \quad e_1 \text{ val}_\Sigma \\
\text{diagap} [U_2^n, V_2^n] (e_1; e_2) & \quad \Sigma \rightarrow \text{diagap} [U_2^n, V_2^n] (e'_1; e'_2)
\end{align*}
\]

\[
\begin{align*}
\text{st-MeasRef} & \quad e @ \Sigma \\
\text{meas} (e) & \quad \Sigma \rightarrow \text{meas} (e')
\end{align*}
\]