The Essence of Q#: Toward Safe and Certified Quantum Programs

(Thesis Proposal)

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

August 23, 2022

Abstract

Q# is one of the only industry-supported standalone quantum programming languages, but it lacks precision in its specification. I propose to provide a formal language definition for Q# through a translation to $\lambda_{Q\#}$, a small core language for quantum computation. This definition leads to a clearer understanding of the Q# language semantics. Using $\lambda_{Q\#}$ as a testbed, I propose the design of a stronger type system that will lead to safer Q# programs. I also propose a verification architecture for $\lambda_{Q\#}$ that can serve as a unified framework for formally specifying, programming, and reasoning about quantum programs.

1 Problem and Motivation

It is difficult to formally specify the definition of a *large* programming language.[1] Even more so when the specification task is done retrospectively—not from the ground up while designing the language. It is no surprise that few programming languages come with a formal language definition.

Why even formalize a language? It is common knowledge in our programming languages (PL) community that sound language design principles lead to programming languages in which programs are easier to write, compose, and maintain. Further, with a formal definition evolving the language with feature extensions is easier. There are several well-known examples, such as Standard ML, Java, JavaScript, Go, and Rust, where formal study and specification have led to greater understanding and extensions (and sometimes removal) of language features (see §4.1 for more).

Quantum computing is a growing discipline where we are bound to face similar problems in software engineering as in the classical domain. Hence,

[1] I use the term "large language" in the sense of Steele [1999].

applying the best lessons learned in PL advances over the last few decades early in quantum computation is essential. A good starting point is the Q# programming language from Microsoft that aims to support the development of large-scale quantum applications while seamlessly integrating classical and quantum computation.

Like most large languages, Q# lacks a formal specification. Written descriptions include a somewhat-dated initial publication by Svore et al. [2018] and the well-maintained *Q# Language Specification* [2020].[2]

These descriptions, being informal, leave a lot to be desired. For example, the *Q# Language Specification* does not say much about what safety properties a programmer can expect from the language. An informal description also makes it harder to analyze the effect of adding or removing language features to the rest of the language.

In this document, I propose to address the following problem statement:

> Provide a formal language definition for Q# through translation to a smaller core language – $\lambda_{Q\#}$. Demonstrate possible extensions to Q#, such as a stronger type system and the ability to produce verified quantum programs using $\lambda_{Q\#}$ as a testbed.

My work will lead to a better understanding of the existing Q# language and help evolve it to support safe and certified quantum programming.

2 Background

2.1 Q# Programming Language

This section summarizes Q# features from the *Q# Language Specification* [2020].

Q# is a standalone domain-specific programming language from Microsoft for writing and running quantum programs. It lets one seamlessly combine classical and quantum computation while providing a separation between pure classical `function`s and effectful quantum `operation`s, collectively known as *callables*. The programming model assumes a high level of abstraction with no notion of quantum circuits or quantum states.

Q# organizes programs using `namespace`s that contain `open` directives, declaration of callables, and type declarations for user-defined types. Namespaces are not hierarchical, however; i.e., there is no relation between two namespaces even if they have a common prefix.

User-defined types declared using `newtype` are essentially records. Callables, as mentioned earlier, include `function`s and `operation`s. Operations may carry *operation characteristics*, `Adj` and `Ctl`, to declare support for *functors*—`Adjoint` and `Controlled`—to enable metaprogramming. They may also include additional *specializations* for their `adjoint` and `controlled` variants, but the reference implementation does not check these specializations for correctness. The default specification is the `body` of the operation.

[2] The latter is considered the authoritative informal definition.

Instead of providing a custom specialization, the user can also ask the compiler to auto-generate a specialization using directives such as `self`, `invert`, and `distribute`. Auto-generation is not possible in all cases, such as when `mutable` variables are involved. If the programmer provides no specialization or generation directive, the directive `auto` applies by default. For example, the following `operation`s are equivalent:

```
operation Trivial() : Unit
is Adj + Ctl { }

operation Trivial() : Unit {
    body (...) { } // dots indicate same arguments as declaration
    adjoint auto;
    controlled auto;
    controlled adjoint auto;
}
```

Q# supports single-line comments that start with double forward slash characters (`//`). There is also support for documentation comments that start with triple slashes (`///`), which may include headers such as *Description*, *Input*, *Output*, *Type Parameters*, *Example*, among others (and are helpful to us, see §3.1).

Statements Statements familiar from classical programming include `return`, `fail`, `let` & `mutable` for variable declaration, `set` for rebinding mutable variables, an iterator-based `for` loop, a conditional `while` loop, and a branching `if-elif-else` conditional. Quantum-specific statements include `use` & `borrow` for qubit allocation, a `repeat-until-fixup` loop for expressing the *repeat-until-success* pattern [Paetznick and Svore 2014], and a `within-apply` statement for expressing the (automatic) *uncomputation* pattern. Statements also include calls to any callable returning `Unit`.

Note that any classical statements may appear in `operation`s but not vice versa for quantum-specific statements and `function`s—a consequence of this restriction is that `function`s can never call `operation`s. Further, the `while` loop is currently not supported inside an `operation`.

The `use` statement provides access to freshly allocated qubits corresponding to state $|0\rangle$, while the `borrow` statement allows access to previously allocated (and potentially entangled) qubits.[3] Q# assumes that a program always starts with no qubits, `operation`s allocate qubits as needed, and they get automatically deallocated at the end of lexical scope, i.e., the lifetime of a qubit variable is equal to its lexical scope.

Expressions Q# expressions are values (literals) and identifiers. They can be combined using *operators*, *modifiers*, and *combinators*. Since all compound data types in Q# are immutable, it includes a notion of a *copy-and-update* operator (`w/ <-`) apart from several familiar operators such as a ternary conditional

[3] I do not consider borrowing in this work because it is an optimization concern that lets a programmer reuse ancillae in their code.

(`? |`), logical, comparison, bitwise, and arithmetic operators. There is also a concatenation operator (+) for `String`s and arrays.

Modifiers include the `Adjoint` and `Controlled` functor applications and the unwrap modifier (`!`) that deconstructs a user-defined type (UDT). Combinators include call (`( )`), named item access (`::`) for UDTs, and array item access (`[ ]`).

Literals include values for most data types: `Unit` (`()`), `Int`, `BigInt`, `Double`, `Bool` (`true` and `false`), `String`, `Result` (`Zero` and `One`), `Pauli` (`PauliI`, `PauliX`, `PauliY`, and `PauliZ`), `Range`, arrays, and tuples. `Qubit` type is opaque and has no literal.

For `function`s and `operation`s, there is support for creating *closures* using *lambda expressions* (`->` and `=>`, respectively) and *partial applications*.

Type System Q# currently features a fairly simple type system. Apart from the primitive data types listed in the preceding section, there is support for compound data types such as arrays, tuples, user-defined types, and callables. Callables are first-class values, and hence, Q# supports higher-order programming. The type of an `operation` follows the schema `<TIn> => <TOut> is <Char>` where `Char`, the operation characteristics (`Adj`, `Ctl`, or both), may be omitted if unneeded. Similarly, the type schema of a `function` is `<TIn> -> <TOut>`. Further, callables allow type parameterization to support generic programming.

There is limited support for subtyping. Specifically, an operation that supports more functors (declared using the operation's characteristics) than needed at the call site may be substituted. Callables require explicit type declarations, but most types in callable bodies are inferrable by Q#'s Hindley-Milner-style type inference algorithm.

Remarks In essence, a Q# program describes quantum computation at a high abstraction level without adherence to the circuit model. With choices such as immutability by default and a clean separation between operations that perform (quantum) side effects and functions that perform pure classical computation, Q# represents a safe synthesis of functional and imperative programming styles in the design space of programming languages for quantum computation. This synthesis is reminiscent of ALGOL-like languages in the classical PL literature. However, there are limitations (see Figure 1). Q# allows uncontrolled aliasing of qubits that, like in classical programming languages, can lead to unsafe computation. Even though the quantum memory model in Q# is to allocate and deallocate qubits in a stack-like manner, the compiler does not enforce this stack discipline.

2.2 $\lambda_{Q\#}$: Q# as a Quantum Algol

When introducing Q#, its language designers, Svore et al. [2018], stated that as opposed to several existing *circuit definition* languages, "Q# is an *algorithm definition* language". In Singhal et al. [2022], along with co-authors, I showed that, in its essence, Q# is a quantum *algorithmic language* (ALGOL).

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

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

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

(b) Using a single qubit as both control and target.

Figure 1: Sample unsafe Q# programs.

- To support this characterization, we introduced $\lambda_{Q\#}$, an idealized version of Q# inspired by Harper's [2016a] language **MA** (Modernized Algol). 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 developed a type system for $\lambda_{Q\#}$ that extends the array-free fragment of Q#'s type system to enforce the no-cloning theorem [Wootters and Zurek 1982] and stack-like management of qubits.
- We provided a dynamic (equational) semantics for $\lambda_{Q\#}$ building upon the fully complete equational theory of quantum computation by Staton [2015].
- Finally, we provided an elaboration relation from Q# to $\lambda_{Q\#}$, thereby endowing a significant portion of Q# with a formal specification and additional safety guarantees.

The $\lambda_{Q\#}$ type system rules out programs equivalent to unsafe Q# programs. Figure 1a shows a minimal example that compiles successfully but fails at runtime (in a simulator) because the implicit stack discipline is currently not enforced by the Q# compiler. In Figure 1b, both `q1` and `q2` refer to the same qubit. Applying `CNOT` operation with `q1` as the control and `q2` as the target is equivalent to cloning the underlying qubit.

In $\lambda_{Q\#}$, neither of these issues arise. The critical idea in $\lambda_{Q\#}$ to enforce stack discipline is to provide the qubit allocation command with its own binding form so that qubits can never escape their lexical scope, i.e., the $\lambda_{Q\#}$ $\texttt{newqref}(x\,.\,m)$ command allocates a new qubit, binds a reference to that qubit to a fresh variable x and makes the variable x available only inside the command m. Once m finishes execution, the qubit referred to by x is automatically deallocated, ensuring safe and simple qubit memory management.

The solution to preventing cloning, i.e., aliasing of qubit references, is inspired by alias types [Smith, Walker, and Morrisett 2000],[4] particularly singleton

[4]Thanks to Mike Hicks for pointing me to alias types.

$\boxed{\Gamma \vdash_\Sigma m \div \tau}$ *(Command m that returns a value of type τ is well-formed wrt Σ)*

$$\textsc{cmd-NewQRef}\ \frac{\Gamma, x : \texttt{qref}\,\langle q\rangle \vdash_{\Sigma,q} m \ \div\ \tau}{\Gamma \vdash_\Sigma \texttt{newqref}\,(x.m) \ \div\ \tau} \qquad \textsc{cmd-GateApRef}\ \frac{\Gamma \vdash_\Sigma e : \texttt{prod}\,(\overline{i \hookrightarrow \texttt{qref}\,\langle q_i\rangle}^{\,i\in 1..n})}{\Gamma \vdash_\Sigma \texttt{gateap}\,\langle U_{2^n}\rangle(e) \ \div\ \texttt{unit}}$$

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

Figure 2: Typing of commands. Γ is the standard typing context. The signature, Σ, keeps track of qubit symbols in scope. Each qubit symbol is distinct.

types [Aspinall 1995; Hayashi 1994]. Using a singleton type indexed by a static symbol, we can easily keep track of logical qubits. In $\lambda_{Q\#}$, a qubit reference, equivalent to `Qubit` in Q#, is typed as $\texttt{qref}\,\langle q\rangle$, where q is a symbol tracked by the type system in the signature, Σ. All variables that refer to the same qubit have the same type; hence, we can statically distinguish references to different qubits. Figure 2 shows the essential rules of the $\lambda_{Q\#}$ type system.

This formal work on $\lambda_{Q\#}$, taking inspiration from the rich literature around ALGOL-like languages [O'Hearn and Tennent 1997a,b], will form the foundation of my dissertation. This version of $\lambda_{Q\#}$ does not cover some interesting Q# features such as arrays, type polymorphism, functor applications, and language-supported abstractions such as `within-apply` and `repeat-until` patterns, which I plan to address in the proposed work.

3 Proposed Work

In this section, I describe two ongoing projects. The first one extends $\lambda_{Q\#}$ to take the project described in §2.2 to its logical conclusion by *a*) enforcing no-cloning for Q# arrays, and *b*) formally describing remaining Q# features. The second project provides a path toward statically enforcing quantum-specific properties (for which static typing techniques are insufficient) using advances in both classical and quantum program verification, thus guaranteeing correct quantum computation before any execution occurs. I conclude with additional directions for future work.

Table 1 shows a rough timeline of my research and graduation plan.

3.1 Extended $\lambda_{Q\#}$: Safe Q# Arrays and More

A formal specification of the whole Q# language requires more work than currently considered in $\lambda_{Q\#}$. Some extensions are straightforward; e.g., we

TABLE 1 A rough research plan

Sept 6, 2022	Candidacy Exam
Sep–Oct '22	Finish extended $\lambda_{Q\#}$ project and write paper
Oct '22	Start job search
Nov 10, '22	Submit extended $\lambda_{Q\#}$ paper to PLDI
Nov–Dec '22	Continue work on certified Q# project
Dec '22	Finish job search
Winter 2023	Write & submit certified Q# paper to ITP/ICFP
Spring '23	Dissertation writing
Spring/Summer '23	Thesis defense

can model classical mutable bindings in Q# after assignables in Harper's **MA**; conveniently, they follow the model of classical local store analogous to how I modeled the quantum local store in Singhal et al. [2022]. Here it helps that Q# does not have a notion of references or pointers and follows the so-called value semantics [Heim et al. 2020].[5] Other features are more challenging, including arrays, slices, iteration, and patterns like `within-apply` and *repeat-until-success*.

A primary goal of $\lambda_{Q\#}$ 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 `Qubit`s to support user-friendly features such as qubit arrays. While convenient, reasoning about interference freedom for arrays is notoriously tricky; specifically, our current approach to enforce no-cloning inspired by alias types [Smith, Walker, and Morrisett 2000; Walker and Morrisett 2001] does not scale to arrays [Walker 2001, §3.5.1]. The core issue is that arrays require a common base type, but with $\texttt{qref}\langle q_i \rangle$ type assigned to each entry of the array, we end up treating the qubit array like a tuple, breaking its abstraction as an array. A solution to this problem is to extend the $\lambda_{Q\#}$ type checker with a constraint solver or offload constraints to a general solver such as Z3 [Moura and Bjørner 2008]. The language designers seem open to changes in the language abstractions if they simplify reasoning and lead to fewer bugs [Geller 2022].

Whiley [Pearce and Groves 2013, 2015] is a modern language that takes the approach of integrating solvers for refined type checking. Whiley recently gained more automation by switching from using its native solver to the industry-scale Z3 SMT solver via a translation to the Boogie intermediate verification language [Barnett et al. 2006; Leino 2008].[6] In their report documenting this effort, Pearce, Utting, and Groves [2022] note that "Wy2B/Boogie/Z3 stack offers significant advantages over the native Whiley verifier in terms of the percentage

[5] Q# treats its `Qubit` type as opaque. In contrast, $\lambda_{Q\#}$ exposes its nature as a reference type explaining why the aliasing problem exists in Q#, even with its value semantics.

[6] Nik Swamy suggested looking into Boogie when I was considering using Z3 directly.

of programs that can be verified automatically."

Along with Jacob Zweifler, I am exploring an architecture similar to Whiley for $\lambda_{Q\#}$.[7] We are extending $\lambda_{Q\#}$ syntax with support for pre- and postconditions for array programs, writing a translation to Boogie, and evaluating how much of the implicit specifications in Q# Arrays library we can encode using our prototype and automatically verify using Boogie/Z3. We have completed an initial prototype for ensuring no-cloning for Q#-like arrays in Whiley and are working on porting it to $\lambda_{Q\#}$.

Here is a motivating example from the Q# standard library:

```
operation ApplyCNOTChain(qs : Qubit[])
: Unit is Adj + Ctl {
    ApplyToEach(CNOT, Zipped(Most(qs), Rest(qs)));
}
```

This program applies the `CNOT` operation to adjacent qubit pairs (q_1, q_2), $(q_2, q_3), \ldots, (q_{n-1}, q_n)$. The type signatures of each of the Q# library callables used above are as follows (`'T` is Q# syntax for type parameters for writing generic callables):

```
function Most<'T> (array : 'T[]) : 'T[]
function Rest<'T> (array : 'T[]) : 'T[]
function Zipped<'T, 'U> (left : 'T[], right : 'U[])
: ('T, 'U)[]
operation CNOT (control : Qubit, target : Qubit)
: Unit is Adj + Ctl
operation ApplyToEach<'T> (op : ('T => Unit), reg : 'T[])
: Unit
```

`Most` returns a new array after dropping the last element, while `Rest` returns a new array after dropping the first element from the input array. `Zipped` returns a new array of pairs of elements from the two input arrays. `ApplyToEach` applies a single-element operation to each register element. Since `CNOT` is a two-element operation, the library authors use an elegant, functional programming idiom to construct the right type of arguments to provide to `ApplyToEach`.

The above program demonstrates both the expressiveness of Q# in combining pure and stateful computation and the challenges in ensuring it is safe to do so. To safely type `ApplyCNOTChain`, the `Qubit` array `qs` requires (as the programmer intends) that its elements are unique, but there is no way in Q# to express that.

A more significant issue is that the implicit specifications associated with the Array library callables are not expressible in the language. However, they do exist in the form of documentation comments (§2.1).[8] Essentially, our solution is to make those specifications explicit in $\lambda_{Q\#}$. Below we see specifications for the example program encoded in Whiley. In the prototype, we specialize all type

[7] Dafny [Leino 2010, 2017] pioneered this architecture. I chose Whiley over Dafny for two reasons: I learned about Whiley before coming across Dafny; and, unlike Dafny, Whiley was developed independently of Boogie providing additional evidence that utilizing Boogie from an unrelated frontend, such as $\lambda_{Q\#}$, could be worthwhile.

[8] See `Arrays/Arrays.qs` and `Arrays/Zip.qs` within the Q# Standard Libraries.

parameters and treat the `Qubit` type as a positive integer because qubit references are essentially integer addresses. Note that there is no direct manipulation of qubits in this program, except through `CNOT` that modifies the quantum state referred to by its inputs.

```
type qbit       is (int n)          where n >= 0
type safePair   is {qbit p, qbit q} where p != q
type distinctOp is method(safePair) -> null

method CNOT(safePair xy) -> null

method ApplyToEach(distinctOp op, safePair[] reg) -> null

function Most(qbit[] qs) -> (qbit[] out)
  requires |qs| > 0
  ensures  |out| == |qs| - 1
  ensures  all { i in 0..|out| | out[i] == qs[i] }

function Rest(qbit[] qs) -> (qbit[] out)
  requires |qs| > 0
  ensures  |out| == |qs| - 1
  ensures  all { i in 0..|out| | out[i] == qs[i + 1] }

function Zipped(qbit[] xs, qbit[] ys) -> (safePair[] xys)
  requires |xs| == |ys|
  requires all { i in 0..|xs| | xs[i] != ys[i] }
  ensures  |xys| == |xs|
  ensures  all { i in 0..|xys| | xys[i].p == xs[i]}
  ensures  all { j in 0..|xys| | xys[j].q == ys[j]}:

method ApplyCNOTChain(qbit[] qs) -> null
  requires |qs| >= 2
  requires all { i in 0..|qs| | qs[i] == qs[0] + i }
```

With these specifications and fairly obvious implementations (except for `Zipped` that requires specifying loop invariants), the program type checks automatically via a translation to Boogie. Our next challenge is to generalize and port these ideas to our current $\lambda_{Q\#}$ type checker along with a translation to Boogie.

I am also working to include coverage of other interesting features of Q# such as iteration, polymorphism, and patterns like `within-apply` and repeat-until-success. Iteration and slices will benefit greatly from the approach described above for arrays (Q# only allows bounded `for` loops, and not the more general `while` loops inside quantum `operation`s).

Before introducing generic arrays, it did not seem worthwhile to include polymorphism in $\lambda_{Q\#}$ but type parametrization over callables is fairly easy to introduce. The `within-apply` conjugation construct is similar to and will be treated along with the `Adjoint` functor.

3.2 Certified Quantum Programming

The previous project will extend $\lambda_{Q\#}$ with syntax for specifying pre- and post-conditions and, potentially, loop invariants. However, it is limited to reasoning about classical properties such as the uniqueness of references or lengths of arrays. What if we wanted to extend $\lambda_{Q\#}$ to also reason about quantum-specific properties? This second project is a step in that direction.

Q# can auto-generate controlled and adjoint specializations of quantum operations but also allows users to supply manual implementations that take precedence. However, the compiler cannot verify whether the user-supplied implementations are valid. Then, there is the question of verifying whether a qubit can be safely deallocated. The answer depends on whether the qubit is unentangled from the rest of the system. These are just two natural quantum-specific properties that arise in the context of Q#. It will also be nice to do fine-grained reasoning about the quantum state as it is often perplexing for newcomers to quantum programming. Some examples include stating whether two (or more) qubits are entangled or separable and whether qubits are in uniform superposition or in some classical state.

This proposal requires both a way to do scalable reasoning without running into the trap of simulating quantum state and, crucially, a usable specification language intuitive to quantum software engineers. Even with significant progress in formalisms such as quantum Hoare and separation logics (see §4.3), the question of a good specification language has largely been ignored in the literature (except recently by Ying [2022]). I believe the lack of an intuitive specification language has hindered the adoption of reasoning tools in the quantum setting.

3.2.1 Toward an ideal specification language

It is relatively easy to specify pure states using the traditional Dirac notation, but unclear how to do the same for mixed states. One solution is to utilize the quantum information-theoretic notion of *purification* [Nielsen and Chuang 2010, pp. 110–111] to specify pure states associated with quantum variables even if they are in mixed states. John Smolin uses the arguably descriptive phrase "Going to the Church of the Larger Hilbert Space" for this idea [Gottesman and Lo 2000]. In other words, any mixed state can be seen as the partial trace of a pure state defined in a larger Hilbert space.

Unruh [2019] uses this idea to introduce an almost satisfactory specification language with support for expressive predicates that can specify quantum variables in a certain distribution, those that are separable, and those in a classical state; in addition to what is possible to state using the predicates of Birkhoff and von Neumann [1936] Quantum Logic (QL) [See also Ying 2022]. These more expressive predicates are defined using Unruh's notion of *ghost variables*. For example, if we want to specify the state of one half of a Bell pair (a, b) after the other half has been measured, we may write $(a, g) : |\Phi^+\rangle$, where g is a ghost variable, and infer that a is in a uniform distribution. Ghost variables are analogous to existential variables but not true existentials because ghost variables may be

```
operation entangle(a : Qubit, b : Qubit) : Unit is Adj
  (requires a:|0⟩, b:|0⟩)
  (ensures  (a,b):|Φ+⟩)
    { H(a); CNOT(a, b); }

operation encode(m : Qubit, a : Qubit) : (Bool, Bool)
  (requires (m,E):α|0⟩|ψ1⟩ + β|1⟩|ψ2⟩, (a,eb):|Φ+⟩)
  (ensures  class(m,a), (em,E,ea,eb):α|+⟩|ψ1⟩|Φ+⟩ + β|-⟩|ψ2⟩|Ψ+⟩)
    { CNOT(m, a); H(m);
      return (M(m) == One, M(a) == One); }

operation decode(b : Qubit, (m1 : Bool, m2 : Bool)) : Unit
  (requires (em,E,ea,b):α|+⟩|ψ1⟩|Φ+⟩ + β|-⟩|ψ2⟩|Ψ+⟩)
  (ensures  (b,E):α|0⟩|ψ1⟩ + β|1⟩|ψ2⟩)
    { if m1 then Z(b); if m2 then X(b); }

operation teleport(m : Qubit, b : Qubit) : Unit
  (requires (m,E):|ψ⟩, b:|0⟩)
  (ensures  class(m), (b,E):|ψ⟩)
    { use a = Qubit();
      entangle(a, b);
      let resultbits = encode(m, a);
      decode(b, resultbits); }
```

Figure 3: Modular teleportation with the specification of its parts

entangled with other quantum variables. We may consider a shift of view, where quantum variables represent subsystems instead of denoting quantum states.[9] With this interpretation, we can say that there exists a quantum subsystem e (following purification), such that the joint system (a, e) is in the Bell state, i.e., $\exists e.(a, e) : |\Phi^+\rangle$. With true existential variables, developing a program logic for reasoning about quantum computation seems feasible. We may even use these existential variables to perform modular and local reasoning over quantum programs that involve entangled states.

Consider a teleportation program in Figure 3 shown in Q# syntax extended with pre- and postconditions, which can transport not just a pure but also an entangled state. Here, qubit m (for message) can be part of a much larger system; everything except the message is represented by an existential variable E, which could be empty in the case of a pure state. a denotes Alice's qubit, b denotes Bob's, and we omit explicit existential quantifiers for conciseness but use the convention that a subscript encodes the relationship between the purifying system and the measurement outcome. Note that we also use existential variables (e.g., e_b in `encode`) to denote part of an entangled system that may not be in

[9]This view shift is easy in the context of Q# and $\lambda_{Q\#}$ since they have no notion of a quantum state.

scope. Further, in the specification of `encode`, we decompose the entangled state $|\psi\rangle$ of the subsystem containing qubit m over the computational basis, where $|\Psi^+\rangle = \frac{|01\rangle+|10\rangle}{\sqrt{2}}$, to allow a type of local reasoning over unentangled states in superposition.

The specification for `teleport` fully specifies what we would like the program to do—we can state that the message qubit is measured (i.e., in a classical state, `class(m)`) at the end and that the message qubit gets swapped by Bob's qubit in the original entangled system. Similarly, in the case of `encode`, we state that we expect qubit a to be a part of a Bell pair and that both input qubits get measured at the end. Meanwhile, existential variables let us keep track of the complete information that may get lost during measurement. The substitution step for existential variables in the `teleport` function after the calls to `encode` and `decode` operations does most of the heavy lifting.

3.2.2 Scalable reasoning using path-sums

The question of scale remains. The specification language described in the previous section is heavily inspired by Unruh's [2019] Hoare-like logic that itself builds upon subspace (or projector) semantics from the quantum logic of Birkhoff and von Neumann [1936]. Recent work has made progress in using projection operators for verification (see §4.3 for more), but apart from scaling challenges, they remain focused on research-like small languages instead of a full-fledged large language like Q#.

An alternative semantics based on Feynman's [1948] path integrals proposed by Amy [2019a,b] called *path-sum semantics* (also known as *sum-over-paths* in the quantum information literature) has shown great progress in performing functional verification over quantum circuits composed of hundreds of qubits. A later proposal by Chareton et al. [2021], *parametrized path-sums* (or PPS), adds support for reasoning over families of circuits. However, both the formalism and its extension seem only suitable for circuits that do not involve measurements.

Lack of measurement may not be a problem, if we can combine the specification language from §3.2.1 with Amy's [2019a,b] path-sum semantics. Another challenge is to figure out how to combine the path-sum formalism designed for unitary quantum circuits with features (such as classical computation) of a large language like Q#. Additionally, we will need to be careful in integrating classical properties proposed in §3.1 with the quantum properties considered here.

I am exploring this idea along with Matt Amy. This work is in an early stage.

3.3 Additional Ideas and Directions

Recently, Microsoft announced Quantum Intermediate Representation [Geller 2020] based on the popular LLVM framework, which has gained significant industry backing in the form of the QIR Alliance [*QIR Specification* 2021]. This development provides an exciting avenue for future research. It will be nice to see a verified compilation stack that starts from a high-level language such as Q# and compiles down (while preserving semantics) to a low-level representation

like QIR. This project will require formally specifying the semantics of QIR, for which we can draw upon the Verified LLVM (Vellvm) project [Zhao et al. 2012].

We can gain confidence in the formalization of $\lambda_{Q\#}$ by mechanizing its metatheory. We see potential in recent developments such as the Agda-based formalization of Second-Order Abstract Syntax [Fiore and Szamozvancev 2022], 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 [Pfenning and Schürmann 1999], Hybrid [Felty and Momigliano 2012], and Beluga [Pientka and Dunfield 2010], 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 [Aydemir et al. 2008; Aydemir and Weirich 2010] that automate the generation of hundreds of lemmas in Coq.

Another way we can make our formalization more rigorous is by taking the tested-semantics approach pioneered by the λ_{JS} project [Guha, Saftoiu, and Krishnamurthi 2010], where the authors compare the outputs generated by the programs written in the surface grammar against those produced by the core after translation. With access to a large corpus of open-source Q# code in the form of Microsoft's quantum libraries and katas [Mykhailova 2020], we are optimistic that this approach will yield results.

4 Related Work

4.1 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 [Harper and Stone 2000; Lee, Crary, and Harper 2007], Java [Igarashi, Pierce, and Wadler 2001], JavaScript [Guha, Saftoiu, and Krishnamurthi 2010], Rust [Jung 2020; Jung et al. 2017], and, most recently, Go [Griesemer et al. 2020]. In this work, we follow the pioneering methodology of the formalization and mechanization of the definition of Standard ML [Milner et al. 1997] by identifying a well-founded core language ($\lambda_{Q\#}$) and performing all metatheoretical reasoning on that core. The above projects demonstrate the extent to which it is possible to distill large and complex languages into their formal and faithful essence. Java and Go are examples of industry-scale languages in mass use benefiting from formalization and academic study: Investigation of extensions such as generics (polymorphism) on languages' cores led to their production adoption over the years. In the case of JavaScript, perhaps the impact of a careful formal study was even more significant as JavaScript is the de-facto programming language 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. 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. We hope our work serves as a similar playground for extensions and future impact.

4.2 Linear & Monadic Quantum Languages

Research-oriented languages like QWIRE [Paykin, Rand, and Zdancewic 2017] and Silq [Bichsel et al. 2020] employ a linear type system to enforce the no-cloning theorem. So far, industry languages, including Q#, have not adopted linear typing. The monadic treatment of state justifies the lack of linear typing in Q#. The monad interface imposes a sequential order to manipulate the quantum state as every monad can be treated as a linear-use state monad [Møgelberg and Staton 2014]. 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 the $\lambda_{Q\#}$ type system addresses.

Other monadic languages include Quantum IO Monad [Altenkirch and Green 2009] (QIO) and Quantum Hoare Type Theory [Singhal 2020; Singhal and Reppy 2021] (QHTT). QIO is a pioneering monadic interface that isolates quantum effects inside a monad as we do in $\lambda_{Q\#}$. QHTT is a typed framework that extends the QIO monad with pre- and postconditions so that precise specifications about the quantum state can be stated and proved in a dependent type theory. My experience developing QHTT motivates the second project proposed in §3.2.

4.3 Quantum Logics and Specification Languages

I discuss important quantum-specific logics and their specification languages here. These logics are designed for artificial research languages and cannot be trivially applied to a large quantum programming language like Q#.

Quantum Hoare Logics Ying's [2012] Quantum Hoare Logic (QHL) builds upon previous work on modeling quantum programs as superoperators [Selinger 2004] and quantum predicates as Hermitian operators [D'Hondt and Panangaden 2006]. QHL and its variants suffer from two problems: First, they require verbose specifications over the complete quantum state because they do not support Separation logic-style frame rules [Reynolds 2002]. Second, the verification process is manual, non-trivial, and does not easily scale to more complex algorithms such as Shor's [1994] as Liu et al. [2019a, §5.2] note while presenting their QHL implementation in an interactive theorem prover. A fundamental reason behind this second problem is the representation of quantum propositions as Hermitian operators.

Indeed, because of this scalability limitation, there is a trend in QHL research to move away from general Hermitian operators and toward a special class of propositions based on projection operators [Li et al. 2020; Yan, Jiang, and Yu 2022; Zhou, Yu, and Ying 2019] rooted in the Quantum Logic (QL) of Birkhoff and von Neumann [1936]. A major limitation of QL is that there is no way to express

probabilistic predicates corresponding to mixed states. Unruh [2019] extends QL using a novel notion of ghost variables so that fine-grained specifications can be given for even mixed states. In my view, Unruh's specification language stands out as state of art in the design of an ideal specification language for reasoning about quantum computation.

In "Quantum Hoare Type Theory" [Singhal 2020; Singhal and Reppy 2021], I proposed a dependently-typed quantum programming language, QHTT, with the capability to perform specification, writing, and verification of quantum programs in a single framework building upon a Hoare Type Theory (HTT) [Nanevski, Morrisett, and Birkedal 2008] in the classical setting. For this work, I used Unruh's [2019] specification language. The missing piece in this work was how to automate this kind of reasoning in a scalable manner. A potential solution is to use Amy's [2019a,b] proposal of path sum semantics, which is what I have proposed (§3.2.2) and which has already shown promising results in the QBricks verification framework [Chareton et al. 2021].

Separation Logics Another approach to overcome the scalability concern has led to a recent surge of interest in separation logics for quantum computation. However, unlike Reynolds' initial attempts to formulate a separation logic for reasoning about *pointer manipulation* using small, intuitive programs, the attempts in the quantum setting have provided precise formulations of theory without providing much intuition.

Zhou et al. [2021] focus on the quantum **while**-language [Ying 2012] that supports neither allocation nor deallocation of qubits and assumes a fixed set of qubits (i.e., no memory management). From the classical viewpoint, the utility of separation logic in this setting feels ill-motivated. The quantum language considered by Le et al. [2022] includes a heap-like model of qubit allocation and deallocation and hence is closer to the traditional problems considered in the classical literature. There is no aliasing, however. Further, because it only supports reasoning about programs that defer measurement to the end, it has limited application potential for general quantum computation.

Neither of these logics provides a clear quantum memory model that quantum software engineers may rely on, nor do they offer a usable specification language to write true statements about the quantum state. The lack of a conceptual quantum memory model and, more importantly, a useful assertion language hampers the scalability of pre- and postconditions, rendering these logics challenging to use in practice. In Singhal, Rand, and Amy [2022], we raised questions about the utility of separation logics in quantum computation and proposed steps toward a user-friendly specification language (as described in §3.2.1) based on my experience in QHTT and Unruh's [2019] work.

5 Dissertation Outline

Here I outline the chapters of my planned dissertation tentatively titled "The Essence of Q#: Toward Safe and Certified Quantum Programs."

1. Introduction
2. Background
3. $\lambda_{Q\#}$: A Core-Calculus for Q#
   - 3.1 Syntax
   - 3.2 Statics
   - 3.3 Dynamics
4. Extending Q# for Safe Arrays
5. Certified Quantum Programming
6. Conclusion

Note that chapter 3 will combine work from Singhal et al. [2022] and that proposed in §3.1 to formally define Q#, while the main focus of chapters 4–5 will be on proposed extensions for Q#: specifying and proving classical properties about Q# arrays (§3.1) in chapter 4 and reasoning about quantum-specific properties (§3.2) in chapter 5.

References

Thorsten Altenkirch and Alexander S. Green. 2009. "The Quantum IO Monad." In: *Semantic Techniques in Quantum Computation*. Ed. by Simon J. Gay and Ian Mackie. Cambridge University Press, Cambridge, UK, 173–205. DOI: 10.1017/CBO9781139193313.006. URL: https://www.cs.nott.ac.uk/~pszt xa/g5xnsc/chapter.pdf (cit. on p. 14).

Matthew Amy. 2019a. "Formal Methods in Quantum Circuit Design." Ph.D. Dissertation. University of Waterloo, Waterloo, Ontario, Canada. URL: http://hdl.handle.net/10012/14480 (cit. on pp. 12, 15).

Matthew Amy. 2019b. "Towards Large-Scale Functional Verification of Universal Quantum Circuits." In: *Proceedings of the 15th International Conference on Quantum Physics and Logic (QPL), Halifax, Canada, June 3–7, 2018* (EPTCS). Vol. 287. Open Publishing Association, Waterloo, NSW, Australia, 1–21. DOI: 10.4204/EPTCS.287.1 (cit. on pp. 12, 15).

David Aspinall. 1995. "Subtyping with Singleton Types." In: *Computer Science Logic* (LNCS). Vol. 933. Springer, 1–15. DOI: 10.1007/BFb0022243. URL: https://homepages.inf.ed.ac.uk/da/papers/lss/ (cit. on p. 6).

Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. "Engineering Formal Metatheory." In: *Proc. POPL '08*. ACM, 3–15. DOI: 10.1145/1328438.1328443. URL: https://github.com/plclub/lngen (cit. on p. 13).

Brian Aydemir and Stephanie Weirich. 2010. *LNgen: Tool Support for Locally Nameless Representations*. Tech. rep. MS-CIS-10-24. University of Pennsylvania Department of Computer and Information Science. URL: https://repository.upenn.edu/cis_reports/933/ (cit. on p. 13).

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2006. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In: *Formal Methods for Components and Objects*. Springer, 364–387. DOI: https://doi.org/10.1007/11804192_17 (cit. on p. 7).

Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. "Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics." In: *Proc. PLDI '20*. ACM, 286–300. DOI: 10.1145/3385412.3386007. URL: https://files.sri.inf.ethz.ch/website/papers/pldi20-silq.pdf (cit. on p. 14).

Garrett Birkhoff and John von Neumann. 1936. "The Logic of Quantum Mechanics." *Ann. Math.*, 37, 4, 823–843. DOI: 10.2307/1968621 (cit. on pp. 10, 12, 14).

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perrelle, and Benoît Valiron. 2021. "An Automated Deductive Verification Framework for Circuit-Building Quantum Programs." In: *Programming Languages and Systems, ESOP 2021* (LNCS). Vol. 12648. Springer, 148–177. DOI: 10.1007/978-3-030-72019-3_6. arXiv: 2003.05841 (cit. on pp. 12, 15).

Ellie D'Hondt and Prakash Panangaden. 2006. "Quantum Weakest Preconditions." *Math. Struct. Comput. Sci.*, 16, 3, 429–451. DOI: 10.1017/S0960129506005251. URL: https://www.cs.mcgill.ca/~prakash/Pubs/weakest_mscs.pdf (cit. on p. 14).

Amy Felty and Alberto Momigliano. 2012. "Hybrid: A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax." *J. Autom. Reason.*, 48, 1, 43–105. DOI: 10.1007/s10817-010-9194-x. arXiv: 0811.4367 (cit. on p. 13).

Richard P. Feynman. 1948. "Space-Time Approach to Non-Relativistic Quantum Mechanics." *Rev. Mod. Phys.*, 20, 367–387, 2. DOI: 10.1103/RevModPhys.20.367. URL: http://www.fafnir.phyast.pitt.edu/py3765/PathIntegral.pdf (cit. on p. 12).

Marcelo Fiore and Dmitrij Szamozvancev. 2022. "Formal Metatheory of Second-Order Abstract Syntax." *Proc. ACM Program. Lang.*, 6, POPL, 53. DOI: 10.1145/3498715 (cit. on p. 13).

Alan Geller. 2020. *Introducing Quantum Intermediate Representation (QIR)*. Q# Blog. URL: https://devblogs.microsoft.com/qsharp/introducing-quantum-intermediate-representation-qir/ (cit. on p. 12).

Alan Geller. 2022. Personal Communication (cit. on p. 7).

Daniel Gottesman and Hoi-Kwong Lo. 2000. "From Quantum Cheating to Quantum Security." *Phys. Today*, 53, 11, 22–27. DOI: 10.1063/1.1333282 (cit. on p. 10).

Robert Griesemer, Raymond Hu, Wen Kokke, Julien Lange, Ian Lance Taylor, Bernardo Toninho, Philip Wadler, and Nobuko Yoshida. 2020. "Featherweight Go." *Proc. ACM Program. Lang.*, 4, OOPSLA, Article 149. DOI: 10.1145/3428217. arXiv: 2005.11710 (cit. on p. 13).

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. 2010. "The Essence of JavaScript." In: *ECOOP 2010 – Object-Oriented Programming*. Springer,

126–150. DOI: 10.1007/978-3-642-14107-2_7. arXiv: 1510.00925 (cit. on p. 13).

Robert Harper. 2016a. "Modernized Algol." In: *Practical Foundations for Programming Languages*. (2nd ed.). Cambridge University Press, Cambridge, UK. Chap. 34, 301–312. DOI: 10.1017/CBO9781316576892.036 (cit. on p. 5).

Robert Harper and Christopher A. Stone. 2000. "A Type-Theoretic Interpretation of Standard ML." In: *Proof, Language, and Interaction: Essays in Honor of Robin Milner*. Ed. by Gordon Plotkin, Colin P. Stirling, and Mads Tofte. MIT Press, Cambridge, MA, 341–387. DOI: 10.7551/mitpress/5641.003.0019. URL: https://www.cs.cmu.edu/~rwh/papers/ttisml/ttisml.pdf (cit. on p. 13).

Susumu Hayashi. 1994. "Singleton, Union, and Intersection Types for Program Extraction." *Inf. Comput.*, 109, 1–2, 174–210. DOI: 10.1006/inco.1994.1016 (cit. on p. 6).

Bettina Heim, Mathias Soeken, Sarah Marshall, C. E. Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta M. Svore. 2020. "Quantum Programming Languages." *Nat. Rev. Phys.*, 2, 709–722. DOI: 10.1038/s42254-020-00245-7 (cit. on p. 7). Also available as Ch. 6–7 of Bettina Heim. "Development of Quantum Applications." Ph.D. Dissertation. ETH Zurich. DOI: 10.3929/ethz-b-000468201.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. "Featherweight Java: A Minimal Core Calculus for Java and GJ." *ACM Trans. Program. Lang. Syst.*, 23, 3, 396–450. DOI: 10.1145/503502.503505 (cit. on p. 13).

Ralf Jung. 2020. "Understanding and Evolving the Rust Programming Language." Ph.D. Dissertation. Saarland University. DOI: 10.22028/D291-31946. URL: https://www.ralfj.de/research/thesis.html (cit. on p. 13).

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. "RustBelt: Securing the Foundations of the Rust Programming Language." *Proc. ACM Program. Lang.*, 2, POPL, Article 66. DOI: 10.1145/3158154 (cit. on p. 13).

Xuan-Bach Le, Shang-Wei Lin, Jun Sun, and David Sanan. 2022. "A Quantum Interpretation of Separating Conjunction for Local Reasoning of Quantum Programs Based on Separation Logic." *Proc. ACM Program. Lang.*, 6, POPL, 36. DOI: 10.1145/3498697 (cit. on p. 15).

Daniel K. Lee, Karl Crary, and Robert Harper. 2007. "Towards a Mechanized Metatheory of Standard ML." In: *Proc. POPL '07*. ACM, 173–184. DOI: 10.1145/1190216.1190245. URL: https://www.cs.cmu.edu/~dklee/papers/tslf-popl.pdf (cit. on p. 13).

K. Rustan M. Leino. 2008. "This is Boogie 2." URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/ (cit. on p. 7).

K. Rustan M. Leino. 2010. "Dafny: An Automatic Program Verifier for Functional Correctness." In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer, 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cit. on p. 8).

K. Rustan M. Leino. 2017. "Accessible Software Verification with Dafny." *IEEE Softw.*, 34, 06, 94–97. DOI: 10.1109/MS.2017.4121212 (cit. on p. 8).

Gushu Li, Li Zhou, Nengkun Yu, Yufei Ding, Mingsheng Ying, and Yuan Xie. 2020. "Projection-Based Runtime Assertions for Testing and Debugging Quantum Programs." *Proc. ACM Program. Lang.*, 4, OOPSLA, 150. DOI: 10.1145/3428218 (cit. on p. 14).

Junyi Liu, Bohua Zhan, Shuling Wang, Shenggang Ying, Tao Liu, Yangjia Li, Mingsheng Ying, and Naijun Zhan. 2019a. "Formal Verification of Quantum Algorithms Using Quantum Hoare Logic." In: *Computer Aided Verification (CAV '19)* (LNCS). Vol. 11562. Springer, 187–207. DOI: 10.1007/978-3-030-25543-5_12 (cit. on p. 14).

Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. 1997. *The Definition of Standard ML.* (Revised ed.). MIT Press, Cambridge, MA. DOI: 10.7551/mitpress/2319.001.0001. URL: https://smlfamily.github.io/sml97-defn.pdf (cit. on p. 13).

Rasmus Ejlers Møgelberg and Sam Staton. 2014. "Linear Usage of State." *Log. Methods Comput. Sci.*, 10, 1, Article 17. DOI: 10.2168/LMCS-10(1:17)2014 (cit. on p. 14).

Leonardo de Moura and Nikolaj Bjørner. 2008. "Z3: An Efficient SMT Solver." In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '08)*. Springer, 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 7).

Mariia Mykhailova. 2020. "The Quantum Katas: Learning Quantum Computing Using Programming Exercises." In: *Proc. SIGCSE 2020*. ACM, 1417. DOI: 10.1145/3328778.3372543. URL: https://github.com/microsoft/QuantumKatas (cit. on p. 13).

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. 2008. "Hoare Type Theory, Polymorphism and Separation." *J. Funct. Program.*, 18, 5-6, 865–911. DOI: 10.1017/S0956796808006953. URL: https://software.imdea.org/~aleks/htt/jfpsep07.pdf (cit. on p. 15).

Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information.* (10th Anniversary Edition ed.). Cambridge University Press, Cambridge. DOI: 10.1017/CBO9780511976667 (cit. on p. 10).

Peter W. O'Hearn and Robert D. Tennent, (Eds.) . 1997a. *ALGOL-like Languages.* Vol. 1. Birkhäuser, Boston, MA. DOI: 10.1007/978-1-4612-4118-8 (cit. on p. 6).

Peter W. O'Hearn and Robert D. Tennent, (Eds.) . 1997b. *ALGOL-like Languages.* Vol. 2. Birkhäuser, Boston, MA. DOI: 10.1007/978-1-4757-3851-3 (cit. on p. 6).

Adam Paetznick and Krysta M. Svore. 2014. "Repeat-Until-Success: Non-deterministic decomposition of single-qubit unitaries." *Quantum Inf. Comput.*, 14, 15 & 16, 1277–1301. DOI: 10.26421/qic14.15-16-2. arXiv: 1311.1074 (cit. on p. 3).

Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. "QWIRE: A Core Language for Quantum Circuits." In: *Proc. POPL '17*. ACM, 846–858. DOI: 10.1145/3009837.3009894. URL: https://jpaykin.github.io/papers/prz_qwire_2017.pdf (cit. on p. 14).

David J. Pearce and Lindsay Groves. 2013. "Whiley: A Platform for Research in Software Verification." In: *Software Language Engineering.* Springer Interna-

tional Publishing, 238–248. DOI: 10.1007/978-3-319-02654-1_13 (cit. on p. 7).

David J. Pearce and Lindsay Groves. 2015. "Designing a verifying compiler: Lessons learned from developing Whiley." *Sci. Comput. Program.*, 113, 191–220. Formal Techniques for Safety-Critical Systems. DOI: https://doi.org/10.1016/j.scico.2015.09.006 (cit. on p. 7).

David J. Pearce, Mark Utting, and Lindsay Groves. 2022. "Verifying Whiley Programs with Boogie." *J. Autom. Reason.* DOI: 10.1007/s10817-022-09619-1 (cit. on p. 7).

Frank Pfenning and Carsten Schürmann. 1999. "System Description: Twelf—A Meta-Logical Framework for Deductive Systems." In: *Automated Deduction—CADE-16*. Springer, 202–206. DOI: 10.1007/3-540-48660-7_14. URL: https://www.cs.cmu.edu/~fp/papers/cade99.pdf (cit. on p. 13).

Brigitte Pientka and Jana Dunfield. 2010. "Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)." In: *Automated Reasoning. IJCAR 2010*. Springer, 15–21. DOI: 10.1007/978-3-642-14203-1_2 (cit. on p. 13).

Q# Language Specification. 2020. Microsoft. URL: https://github.com/microsoft/qsharp-language/tree/main/Specifications/Language#q-language (cit. on p. 2). Orig. Pub. as Ch. 8 of Bettina Heim. "Development of Quantum Applications." Ph.D. Dissertation. ETH Zurich. DOI: 10.3929/ethz-b-000468201.

QIR Specification. 2021. Version 0.1. QIR Alliance: https://qir-alliance.org. URL: https://github.com/qir-alliance/qir-spec (cit. on p. 12).

John C. Reynolds. 2002. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *Proc. LICS '02*. IEEE Computer Society, 55–74. DOI: 10.1109/LICS.2002.1029817. URL: https://www.cs.cmu.edu/~jcr/seplogic.pdf (cit. on p. 14).

Peter Selinger. 2004. "Towards a Quantum Programming Language." *Math. Struct. Comput. Sci.*, 14, 4, 527–586. DOI: 10.1017/S0960129504004256. URL: https://www.mathstat.dal.ca/~selinger/papers/papers/qpl.pdf (cit. on p. 14).

Peter W. Shor. 1994. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring." In: *Proc. FOCS '94*. IEEE Computer Society, 124–134. DOI: 10.1109/SFCS.1994.365700 (cit. on p. 14).

Kartik Singhal. 2020. "Quantum Hoare Type Theory." Master's paper. University of Chicago, Chicago, IL. arXiv: 2012.02154. URL: https://ks.cs.uchicago.edu/publication/qhtt-masters/ (cit. on pp. 14, 15).

Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. 2022. "Q# as a Quantum Algorithmic Language." In: *Proceedings of the 19th International Conference on Quantum Physics and Logic (QPL), Oxford, U.K., June 27–July 1, 2022* (EPTCS). To appear. Open Publishing Association, Waterloo, NSW, Australia. arXiv: 2206.03532. URL: https://ks.cs.uchicago.edu/publication/q-algol/ (cit. on pp. 4, 7, 16).

Kartik Singhal, Robert Rand, and Matt Amy. 2022. "Beyond Separation: Toward a Specification Language for Modular Reasoning about Quantum Programs."

Accepted for poster presentation at PLanQC 2022. URL: https://ks.cs.uchicago.edu/publication/beyond-qsep/ (cit. on p. 15).

Kartik Singhal and John Reppy. 2021. "Quantum Hoare Type Theory: Extended Abstract." In: *Proceedings of the 17th International Conference on Quantum Physics and Logic (QPL), Paris, France, June 2–6, 2020* (EPTCS). Vol. 340. Open Publishing Association, Waterloo, NSW, Australia, 291–302. DOI: 10.4204/EPTCS.340.15. URL: https://ks.cs.uchicago.edu/publication/qhtt/ (cit. on pp. 14, 15).

Frederick Smith, David Walker, and Greg Morrisett. 2000. "Alias Types." In: *Proc. ESOP '00*. Springer, 366–381. DOI: 10.1007/3-540-46425-5_24. URL: https://www.cs.cornell.edu/talc/papers/alias.pdf (cit. on pp. 5, 7).

Sam Staton. 2015. "Algebraic Effects, Linearity, and Quantum Programming Languages." In: *Proc. POPL '15*. ACM, 395–406. DOI: 10.1145/2676726.2676999. URL: http://www.cs.ox.ac.uk/people/samuel.staton/papers/popl2015.pdf (cit. on pp. 5, 13).

Guy L. Steele. 1999. "Growing a Language." *Higher-Order and Symbolic Computation*, 12, 3, 221–236. Invited talk at OOPSLA '98. DOI: 10.1023/A:1010085415024 (cit. on p. 1).

Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, C. E. Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. "Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL." In: *Proc. Real World Domain Specific Languages Workshop (RWDSL) 2018* Article 7. ACM. DOI: 10.1145/3183895.3183901. arXiv: 1803.00652 (cit. on pp. 2, 4).

Dominique Unruh. 2019. "Quantum Hoare Logic with Ghost Variables." In: *Proc. LICS '19*. IEEE Computer Society. DOI: 10.1109/LICS.2019.8785779. arXiv: 1902.00325 (cit. on pp. 10, 12, 15).

David Walker. 2001. "Typed Memory Management." Ph.D. Dissertation. Cornell University. URL: https://www.cs.cmu.edu/~dpw/papers/thesis_ps.gz (cit. on p. 7). Order no. 9988176. An OCR'd copy is available at https://ks.cs.uchicago.edu/files/2001-01-walker-thesis.pdf for convenience.

David Walker and Greg Morrisett. 2001. "Alias Types for Recursive Data Structures." In: *Types in Compilation*. Springer, 177–206. DOI: 10.1007/3-540-45332-6_7. URL: https://www.cs.cornell.edu/talc/papers/alias-recursion.pdf (cit. on p. 7).

W. K. Wootters and W. H. Zurek. 1982. "A single quantum cannot be cloned." *Nature*, 299, 5886, 802–803. DOI: 10.1038/299802a0. URL: https://copilot.caltech.edu/documents/17036/299802a0.pdf (cit. on p. 5).

Peng Yan, Hanru Jiang, and Nengkun Yu. 2022. "On Incorrectness Logic for Quantum Programs." *Proc. ACM Program. Lang.*, 6, OOPSLA1, Article 72. DOI: 10.1145/3527316 (cit. on p. 14).

Mingsheng Ying. 2012. "Floyd–Hoare Logic for Quantum Programs." *ACM Trans. Program. Lang. Syst.*, 33, 6, 19. DOI: 10.1145/2049706.2049708 (cit. on pp. 14, 15).

Mingsheng Ying. 2022. *Birkhoff-von Neumann Quantum Logic as an Assertion Language for Quantum Programs*. arXiv: 2205.01959 (cit. on p. 10).

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations." In: *Proc. POPL '12*. ACM, 427–440. DOI: 10.1145/2103656.2103709. URL: https://www.cis.upenn.edu/~stevez/papers/ZNMZ12.pdf (cit. on p. 13).

Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. "A Quantum Interpretation of Bunched Logic & Quantum Separation Logic." In: *Proc. LICS '21*. IEEE Computer Society, 1–14. DOI: 10.1109/LICS52264.2021.9470673. arXiv: 2102.00329 (cit. on p. 15).

Li Zhou, Nengkun Yu, and Mingsheng Ying. 2019. "An Applied Quantum Hoare Logic." In: *Proc. PLDI '19*. ACM, 1149–1162. DOI: 10.1145/3314221.3314584. URL: https://opus.lib.uts.edu.au/bitstream/10453/140615/2/3314221.3314584.pdf (cit. on p. 14).

Further Reading

John Azariah. 2018. *F# & Q# - A tale of two languages*. F# and Q# Advent Calendars 2018. URL: https://johnazariah.github.io/2018/12/04/tale-of-two-languages.html.

David R. Cok and K. Rustan M. Leino. 2022. "Specifying the Boundary Between Unverified and Verified Code." In: *The Logic of Software. A Tasting Menu of Formal Methods: Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday*. Springer International Publishing, 105–128. DOI: 10.1007/978-3-031-08166-8_6.

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

Alan Geller. 2019. *What are Qubits?* Q# Advent Calendar 2019. URL: https://devblogs.microsoft.com/qsharp/what-are-qubits/.

Robert Harper. 2016b. *Practical Foundations for Programming Languages*. (Second ed.). Abbreviated online edition, with corrections: https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf. Cambridge University Press, Cambridge, UK. DOI: 10.1017/CBO9781316576892.

Bettina Heim. 2020. "Development of Quantum Applications." Ph.D. Dissertation. ETH Zurich. DOI: 10.3929/ethz-b-000468201.

David J. Pearce. 2017. "Array Programming in Whiley." In: *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming* (ARRAY 2017). ACM, 17–24. DOI: 10.1145/3091966.3091972. URL: https://whileydave.com/publications/Pea17_ARRAY_preprint.pdf.

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

Kartik Singhal. 2021. *Quantum Programming Languages & Verification Bibliography*. DOI: 10.5281/zenodo.5780237. URL: https://git.io/qpl-bib.