Q*: Implementing Quantum Separation Logic in F*

KESHA HIETALA, University of Maryland, USA
SARAH MARSHALL, Microsoft Quantum, USA
ROBERT RAND, University of Chicago, USA
NIKHIL SWAMY, Microsoft Research, USA

In this extended abstract we present ongoing work on implementing a quantum separation logic in F* and discuss our prototype implementation of a proof of correctness for teleport with entanglement.

1 OVERVIEW

Recent work [Le et al. 2022; Zhou et al. 2021] has shown that separation logic, used to reason about pointer-manipulating classical programs, is applicable in the domain of quantum computation. Here, the separating conjunction $\star$, traditionally used to describe disjoint portions of the classical heap, describes *separability of quantum states*. A quantum state is *separable* when it can be written as the tensor product of two smaller states, i.e., $\psi = \psi_1 \otimes \psi_2$.[1] This is in contrast to an *entangled* state, which cannot be similarly decomposed. When two quantum bits (*qubits*) are in an entangled state, operations on one qubit may affect the other; so despite the two objects being physically distinct, they must be reasoned about together. On the other hand, if two qubits are in a separable state, then they can be reasoned about independently, allowing for modularity in proofs.

The two current proposals for quantum separation logics [Le et al. 2022; Zhou et al. 2021] do not come equipped with implementations. We are working to rectify this by building a quantum separation logic on top of Steel [Fromherz et al. 2021], a language for developing and verifying concurrent programs embedded in F*. F* is a general-purpose functional programming language with effects and a dependent type system enabling program verification. Its type-checker proves that programs meet their specifications (i.e., satisfy their types) using a combination of SMT solving and interactive proof. After verification, F* programs can be extracted to OCaml, F#, or C. The F* language has been used to certify key internet security protocols, including Transport Layer Security (TLS) [Bhargavan et al. 2017], and produce high-performance cryptographic libraries [Zinzindohoué et al. 2017]. Steel has been used to implement and verify mutable AVL trees, a lock-free version of parallel increment, concurrent queues, and a library for message-passing concurrency.[2]

In order to describe our quantum separation logic, we instantiate Steel's separation logic with a model of quantum state based on a partial commutative monoid whose carrier is a vector of complex numbers and whose operation is the tensor product on vectors. This construction allows us to use the separating conjunction $\star$ to assert properties about separable states, and further, to define actions on quantum states with separation logic specifications. We refer to the extension of F* with quantum actions as Q*.

An action's type stores its return value and pre- and postconditions. We provide the actions listed in Figure 1; return values are shown on the left of the left arrow ($\longleftarrow$) and pre- and postconditions are marked in blue. **alloc** allocates a fresh qubit and returns a reference, **discard** deallocates a qubit, **measure** measures a qubit and returns the result, and **apply** applies a quantum gate. There are two points to note: (i) **discard** requires that its input qubit is not entangled with any other qubits, and (ii) the postcondition of **measure** does not associate output b with a particular probability. Instead, **measure** q chooses a value b (using an external source of randomness) such that the projection

[1] ψ_1 and ψ_2 are defined over disjoint sets of qubits so tensor is commutative, i.e., $\psi_1 \otimes \psi_2 = \psi_2 \otimes \psi_1$.
[2] We could reuse these Steel libraries to develop verified hybrid classical/quantum concurrent programs—but we don't currently have a target application in mind. We welcome suggestions from reviewers or audience members during the talk.

$$\{\,\textbf{emp}\,\}\; q \;\leftarrow\; \textbf{alloc}\,\{\, q \mapsto |0\rangle \,\}$$
$$\{\, q \mapsto |\psi\rangle \,\}\; \textbf{discard}\; q\, \{\,\textbf{emp}\,\}$$
$$\{\, q \cup \overline{q} \mapsto |\psi\rangle \,\}\; b \;\leftarrow\; \textbf{measure}\; q\, \{\, q \mapsto |b\rangle \star \overline{q} \mapsto \mathrm{proj}(q, b, |\psi\rangle) \,\}$$
$$\{\, \overline{q} \mapsto |\psi\rangle \,\}\; \textbf{apply U}\; \overline{q}\, \{\, \overline{q} \mapsto U\, |\psi\rangle \,\}$$

Fig. 1. Pre- and postconditions of Q* actions.

onto the state where q is in state $|b\rangle$ is nonzero. In the postcondition, $\mathrm{proj}(q, b, |\psi\rangle)$ projects out the qubit q, assuming it is in state $|b\rangle$.

The key predicate for reasoning about Q* programs is the "points-to" relation $\overline{q} \mapsto |\psi\rangle$, which says that the set of qubits $\overline{q}$ are in the state $|\psi\rangle$. In the case where $\overline{q}$ is empty, we write **emp**. We adopt the separating conjunction $\star$ from separation logic [Reynolds 2002]; $P_1 \star P_2$ says that we can partition the global quantum state Ψ to produce Ψ_1 and Ψ_2 so that P_1 holds of Ψ_1, P_2 holds of Ψ_2, and $\Psi = \Psi_1 \otimes \Psi_2$. $\star$ is commutative and $P \star \textbf{emp} = P$. The predicate $\overline{q} \mapsto |\psi\rangle$ implies that qubits in $\overline{q}$ are not entangled with qubits in the rest of the program; if they were entangled with some other qubit $q' \notin \overline{q}$, then we would need to write $q' \cup \overline{q} \mapsto |\phi\rangle$ for some $|\phi\rangle$.

When using the separating conjunction $\star$, a key inference rule is the *frame rule*, which says that if we have a proof that program c takes predicate P to Q (i.e., $\{\, P \,\}\; c\; \{\, Q \,\}$), then we can derive that it takes $P \star R$ to $Q \star R$ ($\{\, P \star R \,\}\; c\; \{\, Q \star R \,\}$), assuming no variable occurring free in R is modified by c. This allows us to extend a local specification (like P, Q) to a global one ($P \star R$, $Q \star R$). As an example, say that we have a program that applies a Hadamard H gate to qubit q and we know that q is initially in state $|\psi\rangle$ (i.e., $q \mapsto |\psi\rangle$). After the program executes, we will have that $q \mapsto H\,|\psi\rangle$. Using the frame rule, we can extend this specification to a larger program that also includes qubit q': Say that initially $q' \mapsto |\phi\rangle$, then after the program executes, we can conclude that $q' \mapsto |\phi\rangle \star q \mapsto H\,|\psi\rangle$. In other words, the program acting on q has no effect on q'.

In addition to applying the frame rule, we may also need to manually manipulate the matrix expressions inside predicates and selectively apply the following entailment rule:

$$\overline{q_1} \cup \overline{q_2} \mapsto |\psi_1\rangle_{\overline{q_1}} \otimes |\psi_2\rangle_{\overline{q_2}} \iff (\overline{q_1} \mapsto |\psi_1\rangle) \star (\overline{q_2} \mapsto |\psi_2\rangle).$$

Applying this rule may require reasoning about a state $|\psi\rangle$ to show that it has the form $|\psi_1\rangle_{\overline{q_1}} \otimes |\psi_2\rangle_{\overline{q_2}}$ for some $\overline{q_1}$ and $\overline{q_2}$.

2 EXAMPLE: TELEPORTATION WITH ENTANGLEMENT

The goal of quantum teleportation is to transfer a quantum state from one party (Alice) to another (Bob) using only pre-shared quantum entanglement and two classical bits of information. The protocol uses three qubits: `qA`, which belongs to Alice; `qB`, which belongs to Bob; and `qM`, which contains the state that Alice wants to transfer to Bob. At the end of the protocol, `qA` and `qM` will be in some classical state (having been measured by the protocol) and `qB` will match the original state of `qM`. Importantly, if `qM` was initially entangled with some other set of qubits `qs`, then after the protocol `qB` will end up entangled with `qs` instead. This provides a way to set up entanglement between parties that cannot directly interact, which is a key requirement for the quantum internet [Hermans et al. 2022]. Prior work [Boender et al. 2015; Hietala et al. 2021a; Rand et al. 2018] has verified the teleportation protocol, but not with a general specification allowing `qM` to be entangled with other (external) qubits.

The Q* specification for `teleport` is shown in Listing 1. It takes four inputs, two of which are implicit (marked with `#`). Type `qbit` is a qubit identifier and type `qbits` is a set of type `qbit`. `qvec`

Listing 1. Prettified Q⋆ type for teleportation.

```
val teleport (#qs:qbits)
             (qM:qbit{ disjoint {qM} qs })
             (#st:qvec (union {qM} qs))
             (qB:qbit{ qB <> qM /\ disjoint {qB} qs })
  : STT unit
       (pts_to (union {qM} qs) st `star` pts_to {qB} (ket _ false))
       (fun _ -> pts_to (union {qB} qs) st)
```

$\{\, q_A \mapsto |0\rangle \star q_B \mapsto |0\rangle \,\}$

`Entangle(qAlice, qBob)`

$\{\, \{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \,\}$

$\{\, q_M \cup \overline{q} \mapsto |\psi\rangle \star \{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \,\}$

`let (b1, b2) = SendMsg(qAlice, qMsg)`

$\{\, q_M \mapsto |b_1\rangle \star q_A \mapsto |b_2\rangle \star q_B \cup \overline{q} \mapsto Z_{q_B}^{b_1} X_{q_B}^{b_2} |\psi\rangle \,\}$

$\{\, q_B \cup \overline{q} \mapsto |\phi\rangle \,\}$

`DecodeMsg(qBob, classicalBits);`

$\{\, q_B \cup \overline{q} \mapsto X_{q_B}^{b_2} Z_{q_B}^{b_1} |\phi\rangle \,\}$

(a) Specifications for subroutines

$\{\, q_A \mapsto |0\rangle \star q_B \mapsto |0\rangle \,\}$
$\{\, \{q_A, q_B\} \mapsto |00\rangle \,\}$

`H(qAlice);`

$\{\, \{q_A, q_B\} \mapsto H_{q_A} |00\rangle \,\}$

`CNOT(qAlice, qBob);`

$\{\, \{q_A, q_B\} \mapsto CNOT_{q_A, q_B} H_{q_A} |00\rangle \,\}$
$\{\, \{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \,\}$

(b) Proof sketch for `Entangle`

Fig. 2. Specifications for teleport subroutines. q_A, q_B, and q_M refer to Alice and Bob's initial qubits and the message qubit, respectively. The expression M^b is equal to M if b is true and I (the identity matrix) if b is false. A subscript on a matrix expression, M_q, indicates that the matrix is applied only to qubit q; multiplication extends the matrix to the full dimension via padding.

`qs` is a vector state indexed by the qubit set `qs`. The type `STT a pre post` describes a Steel program with return type `a`. If the initial state satisfies the precondition `pre`, then after execution the state is guaranteed to satisfy postcondition `post`, which is a function over the return value. In the case of `teleport`, the return type is `unit`, so the return value can be ignored. In plain text, the specification in Listing 1 says that if `qM` is initially in some (possibly entangled) state `st` and `qB` is in the state $|0\rangle$, then after the protocol `qB` will be in state `st`.

We summarize the pre- and post-conditions for each component of `teleport` (using the operation names in Listing 2, described below) in Figure 2(a). In Figure 2(b), we sketch the proof that the `Entangle` subroutine matches its specification.

3 LONG-TERM VISION: FORMAL VERIFICATION FOR Q#

A broader goal of our work is to provide formal verification for Microsoft's high-level quantum programming language, Q# [Heim 2020; Svore et al. 2018], and our design of Q⋆ reflects this. Listing 2 shows the implementation of quantum teleportation in Q#. As can be seen, Q# is a hybrid imperative/functional language with a special type for quantum state (`Qubit`) and operations that act on the quantum state (e.g., `H`, `CNOT`, `M`, `use`). Our Q⋆ encoding of the `teleport` program is similar, except that it includes more sophisticated type annotations and intermediate calls to lemmas.

Listing 2. Teleportation in Q#, adapted from the Quantum Katas [Mykhailova 2020]. `Adj` marks an operation as adjointable, and `Adjoint` applies its adjoint.

```
operation Entangle (qAlice : Qubit, qBob : Qubit) : Unit is Adj {
    H(qAlice);
    CNOT(qAlice, qBob);
}

operation SendMsg (qAlice : Qubit, qMsg : Qubit) : (Bool, Bool) {
    Adjoint Entangle(qMsg, qAlice);
    let m1 = M(qMsg);
    let m2 = M(qAlice);
    return (m1 == One, m2 == One);
}

operation DecodeMsg (qBob : Qubit, (b1 : Bool, b2 : Bool)) : Unit {
    if b1 { Z(qBob); }
    if b2 { X(qBob); }
}

operation Teleport (qMsg : Qubit, qBob : Qubit) : Unit {
    use qAlice = Qubit();
    Entangle(qAlice, qBob);
    let classicalBits = SendMsg(qAlice, qMsg);
    DecodeMsg(qBob, classicalBits);
}
```

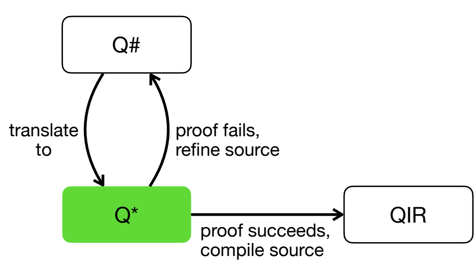

Fig. 3. Overview of (proposed) toolchain for Q#.

Figure 3 presents our vision to integrate Q* into the Q# toolchain: (1) Users write their programs in Q#, taking advantage of the many features available in Microsoft's Quantum Development Kit; (2) The Q# program is translated into Q* and various properties are proved about the Q* representation; (3a) If the proofs succeed, then the original Q# program is compiled to Microsoft's QIR [Geller 2020] or simulated on a classical machine; (3b) If the proofs fail, then the Q# code is refined.

The kinds of properties we might prove about Q* programs include correctness specifications, like the one for `teleport` above, or simpler, Q#-specific well-formedness properties. One such property, naturally enforced by a separation logic, is *discard safety*. It is "safe" to discard a qubit when it is not entangled with any other qubits in the program. In Q#, qubits are implicitly discarded at the end of their lexical scope (e.g., in Listing 2, `qAlice` is discarded at the end of `Teleport`'s body). Discarding an entangled qubit results in an implicit measurement of that qubit, which may change the rest of the program state in unintended ways. To avoid this, the Q# simulator enforces at runtime that discarded qubits are unentangled with the rest of the computation and, additionally, that they are in a classical state. Q⋆'s precondition on **discard** (Figure 1) enforces that the input qubit is unentangled, so proving a property about a program using our separation logic naturally guarantees discard safety.

In some cases, checking for discard safety is easy (e.g., when a qubit is measured before being discarded, or only subject to single-qubit gates), but this is not always the case due to the use of *uncomputation.* It is common practice in quantum computing to use ancilla qubits to store temporary values (e.g., the carry bit in an addition circuit). These ancilla qubits may be entangled with the rest of the state to perform some operation, but they will be uncomputed (and not measured) before the ancilla are discarded. In these cases, proving that a program is discard-safe will require manual reasoning about the underlying vector state.

4 CURRENT STATE

Q⋆ is a work-in-progress. Here we summarize what we have completed so far, and what remains to be done. We invite contributions![3]

- We defined a model of quantum state and its partial commutative monoid.
- Using this model, we implemented actions for generic gate application, discarding a qubit, and ghost operations to share and gather qubits (i.e., the entailment rule from the end of Section 1). Although we have specified types for measurement and allocation, we have yet to finish their implementation.
- We sketched the interface for a F⋆ linear algebra library with a commutative tensor product. Our underlying implementation of matrices and is taken from our previous work on formalizing quantum computing in Coq [Hietala et al. 2021b; Rand et al. 2018]; the key difference is in our implementation of the tensor product. In Coq, we used the Kronecker product, which is not commutative. In Q⋆, we provide the `qvec` type, which is a wrapper around vectors (i.e., single-column matrices) that is indexed by a set of qubits qs. We assume that qubits have some inherent ordering, and we maintain that ordering within the vector. So to implement the tensor product, we use the standard Kronecker product followed by the SWAP operations needed to re-sort the qubits. As a result, if vector v_1 has type `qvec` qs_1 and vector v_2 has type `qvec` qs_2 then $v_1 \otimes v_2 = v_2 \otimes v_1$ and both have type `qvec` $(qs_1 \cup qs_2)$. We have not yet finished the proofs of our implementation of tensor product.
- We implemented the teleport example described above, although our implementation uses the measurement and allocation actions, which are not fully defined, and admitted lemmas related to linear algebra.
- We developed a plugin for the Q# compiler that converts Q# programs into an F⋆ representation [Hietala 2022, Ch. 5], although it requires updates to support the teleport example. For now, we expect users to manually add pre- and postconditions into the generated Q⋆ code, but it is an interesting challenge to consider generating specifications automatically.

ACKNOWLEDGMENTS

Q⋆ grew out of the first author's internship with Microsoft Quantum in Summer 2021, and benefited from conversations with Bettina Heim, Andres Paz, and Cassandra Granade. Q⋆ has also been influenced by joint work with Kartik Singhal on designing $\lambda_{Q\#}$, a small language with a formal semantics describing the core of Q# [Singhal et al. 2022].

[3] https://github.com/microsoft/qsharp-verifier/tree/sep-logic

REFERENCES

Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, Kenji Maillard, Jianyang Pang, Bryan Parno, Jonathan Protzenko, Tahina Ramananandro, Ashay Rane, Aseem Rastogi, Nikhil Swamy, Laure Thompson, Peng Wang, Santiago Zanella-Béguelin, and Jean-Karim Zinzindohoué. 2017. Everest: Towards a Verified, Drop-in Replacement of HTTPS. In *2nd Summit on Advances in Programming Languages*. http://drops.dagstuhl.de/opus/volltexte/2017/7119/pdf/LIPIcs-SNAPL-2017-1.pdf

Jaap Boender, Florian Kammüller, and Rajagopal Nagarajan. 2015. Formalization of Quantum Protocols using Coq. In *Proceedings of the 12th International Workshop on Quantum Physics and Logic (QPL), Oxford, U.K., July 15–17, 2015 (Electronic Proceedings in Theoretical Computer Science, Vol. 195)*, Chris Heunen, Peter Selinger, and Jamie Vicary (Eds.). Open Publishing Association, Waterloo, NSW, Australia, 71–83. https://doi.org/10.4204/EPTCS.195.6

Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. *Proc. ACM Program. Lang.* 5, ICFP, Article 85 (2021), 30 pages. https://doi.org/10.1145/3473590

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

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

S. L. N. Hermans, M. Pompili, H. K. C. Beukers, S. Baier, J. Borregaard, and R. Hanson. 2022. Qubit teleportation between non-neighbouring nodes in a quantum network. *Nature* 605, 7911 (2022), 663–668. https://doi.org/10.1038/s41586-022-04697-y

Kesha Hietala. 2022. *A Verified Software Toolchain For Quantum Programming.* Ph. D. Dissertation. University of Maryland. https://khieta.github.io/files/dissertation-draft.pdf

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021a. Proving Quantum Programs Correct. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 21, 19 pages. https://doi.org/10.4230/LIPIcs.ITP.2021.21

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021b. A Verified Optimizer for Quantum Circuits. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 37 (2021), 29 pages. https://doi.org/10.1145/3434318

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. *Proceedings of the ACM on Programming Languages* 6, POPL, Article 36 (Jan. 2022), 27 pages. https://doi.org/10.1145/3498697

Mariia Mykhailova. 2020. The Quantum Katas: Learning Quantum Computing Using Programming Exercises. In *Proc. SIGCSE 2020.* ACM, New York, NY, USA, 1417. https://doi.org/10.1145/3328778.3372543

Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings of the 14th International Conference on Quantum Physics and Logic (QPL), Nijmegen, the Netherlands, July 3–7, 2017 (Electronic Proceedings in Theoretical Computer Science, Vol. 266)*, Bob Coecke and Aleks Kissinger (Eds.). Open Publishing Association, Waterloo, NSW, Australia, 119–132. https://doi.org/10.4204/EPTCS.266.8

J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* 55–74. https://doi.org/10.1109/LICS.2002.1029817

Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. 2022. Q# as a Quantum Algorithmic Language. https://ks.cs.uchicago.edu/publication/q-algol/

Krysta M. Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher 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 *Proceedings of the Real World Domain Specific Languages Workshop 2018 (RWDSL '18).* Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. https://doi.org/10.1145/3183895.3183901

Li Zhou, Gilles Barthe, Justin Hsu, Mingsheng Ying, and Nengkun Yu. 2021. A Quantum Interpretation of Bunched Logic & Quantum Separation Logic. In *Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '21).* IEEE Computer Society, Los Alamitos, CA, USA, 1–14. https://doi.org/10.1109/LICS52264.2021.9470673

Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17).* Association for Computing Machinery, New York, NY, USA, 1789–1806. https://doi.org/10.1145/3133956.3134043