Q*: Implementing Quantum Separation Logic in F*

Kesha Hietala, Sarah Marshall, Robert Rand, Nikhil Swamy
Amazon               Microsoft              UChicago          Microsoft
Background: Separation Logic

- An extension of Hoare Logic with additional operators including ★ ("separating conjunction"), which describes disjoint parts of the heap

- \{x \mapsto 0 \star y \mapsto 0\} says that variables x and y both have value 0, and that they are distinct (i.e., not aliases of each other)
Background: Separation Logic

• An extension of Hoare Logic with additional operators including ★ ("separating conjunction"), which describes disjoint parts of the heap

• \{ x \mapsto 0 \; \star \; y \mapsto 0 \} says that variables x and y both have value 0, and that they are distinct (i.e., not aliases of each other)

• The frame rule supports scalable reasoning

\[
\frac{\{P\} \; C \; \{Q\}}{\{P \star R\} \; C \; \{Q \star R\}}, \; \text{mod}(C) \cap \text{fv}(R) = \emptyset
\]

• Allows us to prove a “local” property \{P\} \; C \; \{Q\} and extend it to a “global” property \{P \star R\} \; C \; \{Q \star R\}
Quantum Separation Logic?

- ★ describes *separability* of quantum states

- $P_1 \bowtie P_2$ says that we can partition the global state $\Psi$ into $\Psi_1$ and $\Psi_2$ such that $P_1$ holds of $\Psi_1$, $P_2$ holds of $\Psi_2$, and $\Psi = \Psi_1 \otimes \Psi_2$

- Provides a convenient notation for describing whether states are entangled

- Allows us to reason modularly about parts of the state that are not entangled
Quantum Separation Logic?

• ⭐ describes *separability* of quantum states
  
  • $P_1 \star P_2$ says that we can partition the global state $\Psi$ into $\Psi_1$ and $\Psi_2$ such that $P_1$ holds of $\Psi_1$, $P_2$ holds of $\Psi_2$, and $\Psi = \Psi_1 \otimes \Psi_2$

• Provides a convenient notation for describing whether states are entangled

• Allows us to reason modularly about parts of the state that are not entangled

• Proposed by *Zhou at el. (2021)* and *Le et al. (2022)*

• But no implementation
F*: A Proof-Oriented Programming Language

• F* is a functional programming language and proof assistant from Microsoft Research
  • Uses the Z3 solver in the backend for automation

https://www.fstar-lang.org/
F*: A Proof-Oriented Programming Language

- F* is a functional programming language and proof assistant from Microsoft Research
  - Uses the Z3 solver in the backend for automation

- Steel (Fromherz et al. (2021)) is an F* implementation of a concurrent separation logic
  - By building on top of Steel, we get a framework for the ★ operator and frame rule “for free”

https://www.fstar-lang.org/
Modeling Quantum State

• In order to interpret ★ in Steel’s separation logic, we need a model of quantum state & a partial commutative monoid over it
Modeling Quantum State

• In order to interpret ★ in Steel’s separation logic, we need a model of quantum state & a partial commutative monoid over it

• We define a type qvec qs, which is a wrapper around a complex vector of length $2^{|qs|}$ with a commutative definition of tensor

• Our underlying matrix library is a port of the QuantumLib library (used in SQIR and QWIRE) from Coq to F*

• Our commutative definition of tensor is a work-in-progress, but our idea is to apply the standard Kronecker product followed by a permutation matrix to maintain a fixed ordering of qubits
Q* = F* with Quantum Actions

- We introduce a predicate $\overline{q} \mapsto |\psi\rangle$, which says that the set of qubits $\overline{q}$ are collectively in state $|\psi\rangle$ (and, implicitly, unentangled with outside qubits)

$$\begin{align*}
\{ \text{emp} \} q &\leftarrow \text{alloc} \{ q \mapsto |0\rangle \} \\
\{ q \mapsto |\psi\rangle \} \text{discard} q \{ \text{emp} \} \\
\{ q \cup \overline{q} \mapsto |\psi\rangle \} b &\leftarrow \text{measure} q \{ q \mapsto |b\rangle \ast \overline{q} \mapsto \text{proj}(q, b, |\psi\rangle) \} \\
\{ \overline{q} \mapsto |\psi\rangle \} \text{apply} U \overline{q} \{ \overline{q} \mapsto U|\psi\rangle \}
\end{align*}$$
Q* = F* with Quantum Actions

• We introduce a predicate \( \overline{q} \leftrightarrow |\psi\rangle \), which says that the set of qubits \( \overline{q} \) are collectively in state \( |\psi\rangle \) (and, implicitly, unentangled with outside qubits)

• We define four quantum actions

\[
\begin{align*}
\{ \text{emp} \} & q \leftarrow \text{alloc} \{ q \leftrightarrow |0\rangle \} \\
\{ q \leftrightarrow |\psi\rangle \} & \text{discard} q \{ \text{emp} \} \\
\{ q \cup \overline{q} \leftrightarrow |\psi\rangle \} & b \leftarrow \text{measure} q \{ q \leftrightarrow |b\rangle \star \overline{q} \leftrightarrow \text{proj}(q, b, |\psi\rangle) \} \\
\{ \overline{q} \leftrightarrow |\psi\rangle \} & \text{apply} U \overline{q} \{ \overline{q} \leftrightarrow U |\psi\rangle \}
\end{align*}
\]
Q* = F* with Quantum Actions

- We introduce a predicate $\bar{q} \mapsto |\psi\rangle$, which says that the set of qubits $\bar{q}$ are collectively in state $|\psi\rangle$ (and, implicitly, unentangled with outside qubits).

- We define four quantum actions

  \[
  \begin{align*}
  \{ \text{emp} \} & q \leftarrow \text{alloc} \{ q \mapsto |0\rangle \} \\
  \{ q \mapsto |\psi\rangle \} & \text{discard q} \{ \text{emp} \} \\
  \{ q \cup \bar{q} \mapsto |\psi\rangle \} & b \leftarrow \text{measure q} \{ q \mapsto |b\rangle \ast \bar{q} \mapsto \text{proj}(q, b, |\psi\rangle) \} \\
  \{ \bar{q} \mapsto |\psi\rangle \} & \text{apply U q} \{ \bar{q} \mapsto U |\psi\rangle \} 
  \end{align*}
  \]

- And an entailment rule

  $$
  \bar{q}_1 \cup \bar{q}_2 \mapsto |\psi_1\rangle_{q_1} \otimes |\psi_2\rangle_{\bar{q}_2} \iff (\bar{q}_1 \mapsto |\psi_1\rangle) \ast (\bar{q}_2 \mapsto |\psi_2\rangle)
  $$
Example: Quantum Teleportation

```qsharp
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);
}
```
Example: Quantum Teleportation

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);
}
Example: Quantum Teleportation (written in Microsoft’s Q# language)

```qsharp
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);
}
```
Example: Quantum Teleportation

```qsharp
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);
}
```
Example: Quantum Teleportation

```qsharp
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);
}
```
Example: Quantum Teleportation  
(written in Microsoft's Q# language)

```
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);
}
```
**Example: Quantum Teleportation**

\[
\begin{align*}
\{q_A \mapsto |0\rangle & \times q_B \mapsto |0\rangle \} & \text{Entangle}(q_A, q_B) & \{\{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}}( |00\rangle + |11\rangle)\} \\
\{q_M \cup \bar{q} \mapsto |\phi\rangle & \times \{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}}( |00\rangle + |11\rangle)\} & \text{let} & \text{(b1,b2)} = \text{SendMsg}(q_A, q_M) \\
\{q_M \mapsto |b_1\rangle & \times q_A \mapsto |b_2\rangle & \times q_B \cup \bar{q} \mapsto Z_{q_B}^{b_1}X_{q_B}^{b_2} |\phi\rangle\} \\
\{q_B \cup \bar{q} \mapsto |\phi\rangle\} & \text{DecodeMsg}(q_B, (b1,b2)) & \{q_B \cup \bar{q} \mapsto X_{q_B}^{b_2}Z_{q_B}^{b_1} |\phi\rangle\}
\end{align*}
\]
Example: Quantum Teleportation

\[
\begin{align*}
\{q_A \mapsto |0\rangle \} \quad \text{Entangle}(qA, qB) \quad \{\{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)\}
\end{align*}
\]

\[
\begin{align*}
\{q_M \cup \bar{q} \mapsto |\phi\rangle \} \quad \{\{q_A, q_B\} \mapsto \frac{1}{\sqrt{2}} (|00\rangle + |11\rangle)\}
\end{align*}
\]

let \((b_1, b_2) = \text{SendMsg}(qA, qM)\)

\[
\begin{align*}
\{q_M \mapsto |b_1\rangle \} \quad q_A \mapsto |b_2\rangle \quad q_B \cup \bar{q} \mapsto Z_{q_B}^b X_{q_B}^b |\phi\rangle
\end{align*}
\]

\[
\begin{align*}
\{q_B \cup \bar{q} \mapsto |\phi\rangle \} \quad \text{DecodeMsg}(qB, (b_1, b_2)) \quad \{q_B \cup \bar{q} \mapsto X_{q_B}^b Z_{q_B}^b |\phi\rangle\}
\end{align*}
\]

\[
\begin{align*}
\{q_M \cup \bar{q} \mapsto |\phi\rangle \} \quad q_B \mapsto |0\rangle \quad \text{Teleport}(qM, qB) \quad \{q_B \cup \bar{q} \mapsto |\phi\rangle\}
\end{align*}
\]
Prototype Implementation

• Available at github.com/microsoft/qsharp-verifier/tree/sep-logic
  • Not under active development, but open to contributions!
Prototype Implementation

- Available at github.com/microsoft/qsharp-verifier/tree/sep-logic
- Not under active development, but open to contributions!

- Type for teleport:

```ocaml
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)
```
Prototype Implementation

• Available at [github.com/microsoft/qsharp-verifier/tree/sep-logic](https://github.com/microsoft/qsharp-verifier/tree/sep-logic)

• Not under active development, but open to contributions!

• Type for teleport:

```fsharp
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)
```

“other” qubits in the environment (implicit)
Prototype Implementation

• Available at github.com/microsoft/qsharp-verifier/tree/sep-logic

• Not under active development, but open to contributions!

• Type for teleport:

```fsharp
val teleport (#qs:qbits) : STT unit
  (qM:qbit{ disjoint {qM} qs })
  (#st:qvec (union {qM} qs))
  (qB:qbit{ qB <> qM /\ disjoint {qB} qs })

  (pts_to (union {qM} qs) st `star` pts_to {qB} (ket _ false))
  (fun _ -> pts_to (union {qB} qs) st)
```

“other” qubits in the environment (implicit)
message qubit, distinct from qs
Prototype Implementation

• Available at github.com/microsoft/qsharp-verifier/tree/sep-logic

• Not under active development, but open to contributions!

• Type for teleport:

```
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)
```

“other” qubits in the environment (implicit)
message qubit, distinct from qs
initial state of qM and qs (implicit)
Prototype Implementation

- Available at [github.com/microsoft/qsharp-verifier/tree/sep-logic](https://github.com/microsoft/qsharp-verifier/tree/sep-logic)
- Not under active development, but open to contributions!

- Type for teleport:

```ocaml
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)
```

"other" qubits in the environment (implicit)

message qubit, distinct from qs

initial state of qM and qs (implicit)

Bob's qubit, distinct from qs and qM
Prototype Implementation

• Available at github.com/microsoft/qsharp-verifier/tree/sep-logic

• Not under active development, but open to contributions!

• Type for teleport:

```plaintext
val teleport (#qs:qbits)  
(qM:qbit{ disjoint {qM} qs })  
(#st:qvec (union {qM} qs))  
(qB:qbit{ qB <> qM \ / \ disjoint {qB} qs })  
: STT unit
```

“other” qubits in the environment (implicit)
message qubit, distinct from qs
initial state of qM and qs (implicit)
Bob’s qubit, distinct from qs and qM

Steel return type is unit

(PTS_to (union {qM} qs) st `star` PTS_to {qB} (ket _ false))
(fun _ -> PTS_to (union {qB} qs) st)
Prototype Implementation

- Available at github.com/microsoft/qsharp-verifier/tree/sep-logic

- Not under active development, but open to contributions!

- Type for teleport:

```ocaml
define teleport (#qs: qbits) as
  (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)
```

“other” qubits in the environment (implicit)
message qubit, distinct from qs
initial state of qM and qs (implicit)
Bob’s qubit, distinct from qs and qM
Steel return type is unit
precondition: \{qm \cup \overline{q} \mapsto |\psi\rangle * qb \mapsto |0\rangle \}
Prototype Implementation

- Available at github.com/microsoft/qsharp-verifier/tree/sep-logic
- Not under active development, but open to contributions!

- Type for teleport:

```plaintext
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)
```

Notes:
- `disjoint {qM} qs` specifies a message qubit, distinct from `qs`.
- `initial state of qM and qs (implicit)`
- `Steel return type is unit`
- `precondition: {qM ∪ qB ← |ψ⟩ * qB ← |0⟩}`
- `postcondition: {qB ∪ q ← |ψ⟩}`

“other” qubits in the environment (implicit)
Prototype Implementation

```plaintext
let teleport =
  let qA = alloc () in
  disjointness (single qA) (single qB) #_;
  disjointness (single qA) (union (single qM) qs) #_;
  entangle qA qB;
  let bits = send_msg qA qM #_ in
  decode_msg qB qs bits;
  discard qA _;
  discard qM _;
  teleport_lemma (fst bits) (snd bits) qB qs
    (relabel_indices (union (single qB) qs) state);
  rewrite (pts_to (union (single qB) qs) _) (pts_to (union (single qB) qs)
    (relabel_indices (union (single qB) qs) state))
```

let teleport
= let qA = alloc () in
  disjointness (single qA) (single qB) #_;
  disjointness (single qA) (union (single qM) qs) #_;
  entangle qA qB;
  let bits = send_msg qA qM #_ in
  decode_msg qB qs bits;
  discard qA _;
  discard qM _;
  teleport_lemma (fst bits) (snd bits) qB qs
  (relabel_indices (union (single qB) qs) state);
  rewrite (pts_to (union (single qB) qs) _)
   (pts_to (union (single qB) qs)
        (relabel_indices (union (single qB) qs) state))
Prototype Implementation

let teleport =
  let qA = alloc () in
  disjointness (single qA) (single qB) #_;
  disjointness (single qA) (union (single qM) qs) #_;
  entangle qA qB;
  let bits = send_msg qA qM #_ in
  decode_msg qB qs bits;
  discard qA _;
  discard qM _;
  teleport_lemma (fst bits) (snd bits) qB qs
    (relabel_indices (union (single qB) qs) state);
  rewrite (pts_to (union (single qB) qs) _)
    (pts_to (union (single qB) qs)
      (relabel_indices (union (single qB) qs) state))
Additional Applications

• Discard safety
  • A qubit must be unentangled when deallocated

• Qubit resetting and reuse
  • Confirm that a qubit is in the $|0\rangle$ state on discard

• No cloning
  • Check whether qubits alias one another

Future Directions

• Verify more interesting classical/quantum programs
  • Idea: Use Steel to reason about hybrid quantum/classical concurrent programs
• Fix rough edges in the implementation (many admits in our lin. algebra code)
• Integrate Q* into the Q# toolchain
Future Directions

- Verify more interesting classical/quantum programs
  - Idea: Use Steel to reason about hybrid quantum/classical concurrent programs
- Fix rough edges in the implementation (many admits in our lin. algebra code)
- Integrate Q* into the Q# toolchain

Code available at: github.com/microsoft/qsharp-verifier
Separation logic w/ teleport example in the sep-logic branch