1 Postulates

1.1 Mixed States

In general a quantum mechanical system is described mathematically by an operator ρ acting on a Hilbert space $\mathbb{H}$ with ρ having the properties:

1.1.1 ρ is self-adjoint (Hermitian)

$$\rho^* = \rho\,.$$

1.1.2 ρ is positive-definite

$$\rho \geq 0\,.$$

1.1.3 ρ has trace 1

$$\mathrm{tr}\,(\rho) = 1\,.$$

The operator ρ is called **density operator** and we denote the set of density operators on a Hilbert space $\mathbb{H}$ by $\mathrm{D}\,(\mathbb{H})$, that is,

$$\mathrm{D}\,(\mathbb{H}) := \{\rho \in \mathrm{L}\,(\mathbb{H}) \mid \rho^* = \rho,\, \rho \geq 0,\, \mathrm{tr}\,(\rho) = 1\}$$

The quantum states described by density operators in $\mathrm{D}\,(\mathbb{H})$ are called **mixed states**.

1.2 Observables and States (Expectation Value)

The quantum mechanical **expectation value** of an observable A in a mixed state ρ is given by

$$\langle A \rangle_\rho := \mathrm{tr}\,(\rho A)\ .$$

1.3 Measurement Probability

If the quantum system is in a state ρ, λ is an eigenvalue of A and P_λ the projection onto the eigenspace of λ, then the probability $\mathbf{P}_\rho\,(\lambda)$ that a measurement of A yields the value λ is given by

$$\mathbf{P}_\rho\,(\lambda) = \mathrm{tr}\,(\rho P_\lambda)\ .$$

1.4 Projection Postulate (Measurement)

If the quantum system is initially described by the state ρ, and then the measurement of the observable A yields the eigenvalue λ of A, then this measurement has effected the following change of state

$$\rho = \begin{array}{c}\text{state before}\\ \text{measurement}\end{array} \quad \xrightarrow{\text{measurement}} \quad \frac{P_\lambda \rho P_\lambda}{\mathrm{tr}\,(\rho P_\lambda)} = \frac{P_\lambda \rho P_\lambda}{\mathbf{P}_\rho\,(\lambda)} = \begin{array}{c}\text{state after}\\ \text{measurement}\end{array}$$

where P_λ is the projection onto the eigenspace λ.

1.5 Time Evolution (Unitary operation)

Any time evolution of quantum system that is not caused by a measurement is described as an evolution of states

$$\rho\,(t_0) = \text{state at time } t_0 \quad \xrightarrow{\text{no measurement}} \quad \rho\,(t) = \text{state at time } t$$

gien by a unitary time evolution operator $U\,(t, t_0)$ acting on the density operator as

$$\rho\,(t) = U\,(t, t_0)\,\rho\,(t_0)\,U\,(t, t_0)^*\ .$$

2 Implemenation

2.1 Instructions

The reduced instructions in the Coq implementation are designed to support all the features in OpenQASM2.0, after trivial inlining of syntactic sugars:

```
(* Program.v *)
Inductive Instruction: Type :=
| NopInstr: Instruction
| RotateInstr: R -> R -> R -> nat -> Instruction (* U (theta phi lambda) qbit *)
| CnotInstr: nat -> nat -> Instruction (* CnotInstr a b: flip b iff a *)
| MeasureInstr: nat -> nat -> Instruction (* MeasureInstr q c: *)
| SeqInstr: Instruction -> Instruction -> Instruction
| IfInstr: nat -> bool -> Instruction -> Instruction. (* if cbit == 0 (false) or cbit == 1 (true) *)
```

2.2 Initial State

We have assumed that an initial state of the execution model can be arbitrary sequence of $|0\rangle$ or $|1\rangle$ qubit:

```
(* Density.v *)
Inductive InitialDensityMatrix: nat -> Matrix -> Prop :=
| DensityMatrix_empty: InitialDensityMatrix 0 (eye 0)
| DensityMatrix_0: InitialDensityMatrix 1 Den_0
| DensityMatrix_1: InitialDensityMatrix 1 Den_1
| DensityMatrix_TMproduct (n1 n2: nat) (den1 den2: Matrix):
  InitialDensityMatrix n1 den1 ->
  InitialDensityMatrix n2 den2 ->
  InitialDensityMatrix (n1 + n2) (TMproduct den1 den2).
```

Note that in OpenQASM2.0, qubits are all initialized to $|0\rangle$ by default.

2.3 Many-Worlds Interpretation

The concept of Many-Worlds Interpreterpretation (MWI) in quantum mechanics has been employed to design the program state of the execution model, referred to as `ManyWorld`. `ManyWorld` is a list of `world`, each representing every probable outcome resulting from different measurement.

```
(* Program.v *)
Record World: Type := {
  W_qstate: Matrix; (* density matrix *)
  W_cstate: total_map bool; (* false for 0, true for 1 *)
  W_prob: R; (* probability of the world *)
  W_num_qubits: nat;
  W_qstate_valid: Mbits W_qstate = W_num_qubits
}.

Definition ManyWorld: Type := list World.
```

3 Verification

3.1 Basics

3.1.1 Matrix / Vector Definition

In our execution model implementation, matrices and vectors are represented by functions of complex numbers and their sizes;

```
(* Matrix.v *)
Record Matrix: Type := {
  Mbits: nat;
  Minner: nat -> nat -> C;
}.
Record RowVec: Type := {
  RVbits: nat;
  RVinner: nat -> C;
```

```
}.
Record ColVec: Type := {
  CVbits: nat;
  CVinner: nat -> C;
}.
```

To ensure validity, accessing elements of matrices or vectors is restricted to proper indices:

```
(* Matrix.v *)
Definition Mget (m : Matrix) (i j: nat) (Hi: i < Msize m) (Hj: j < Msize m): C := Minner m i j.
Definition RVget (r : RowVec) (j: nat) (Hj: j < RVsize r): C := RVinner r j.
Definition CVget (c : ColVec) (i: nat) (Hi: i < CVsize c): C := CVinner c i.
```

3.1.2 (Matrix / Vector) Element-Wise (Unary / Binary) Operation / Multiplication

In all matrix operations, we need to validate that the size of each matrix is appropriate for the operation. Below are the basic correctness of unary and binary element-wise operations:

```
(* Matrix.v *)
Lemma Muop_correct: forall
  (uop: C -> C) (m: Matrix) (i j: nat)
  (H1i: _) (H1j: _) (H2i: _) (H2j: _),
  (Muop uop m)[[i H2i|j H2j]] = uop (m[[i H1i|j H1j]]). (* notation of Mget *)
Lemma Mbop_correct: forall
  (bop: C -> C -> C) (m1 m2 m3: Matrix) (i j: nat)
  (Hbits: _) (H1i: _) (H1j: _) (H2i: _) (H2j: _) (H3i: _) (H3j: _),
  (Mbop bop m1 m2 Hbits)[[i H3i|j H3j]] = bop (m1[[i H1i|j H1j]]) (m2[[i H2i|j H2j]]).
```

And here's the basic correctness of matrix multiplications:

```
(* Matrix.v *)
Lemma Mmult_correct: forall (m1 m2 m: Matrix) (r: RowVec) (c: ColVec) (i j: nat)
  (Hi: _) (Hj: _) (H: _) (Hmi: _) (Hmj: _) (Hrc: _),
  (Mmult m1 m2 H)[[i Hmi|j Hmj]] = dot_product (extract_row m1 i Hi) (extract_col m2 j Hj) Hrc.
```

The proofs regarding vector operations are similar as matrix operations.

3.1.3 Matrix / Vector Tensor Product (Kronecker Product)

Here is the basic correctness of tensor products:

```
(* Tensor.v *)
Lemma TMproduct_correct: forall
  (m1 m2 mt: Matrix) (i j: nat) (Hi: _) (Hj: _) (H1i: _) (H1j: _) (H2i: _) (H2j: _),
  (TMproduct m1 m2)[[i Hi|j Hj]] =
  m1[[(i / Msize m2) H1i|(j / Msize m2) H1j]] * m2[[(i mod Msize m2) H2i|(j mod Msize m2) H2j]].
```

Essential properties of tensor products and other operations have also been proven, as demonstrated below:

```
(* Tensor.v *)
Lemma TMproduct_mult: forall
  (m1 m2 m3 m4: Matrix) (H13: _) (H24: _) (H1234: _),
  TMproduct (Mmult m1 m3 H13) (Mmult m2 m4 H24) = Mmult (TMproduct m1 m2) (TMproduct m3 m4) H1234.
```

3.2 QM-Related Functions in the Execution Model

Postulate 2 and 3 describe how to calculate essential values for quantum circuit execution.

3.2.1 Observables and States (Expectation Value)

"Observables and States" postulate details how to calculate expectation values of observables. In the context of quantum computing, observables are σ_z, which corresponds to a measurement in the standard Z basis, or the *"computational"* basis ($|0\rangle$ and $|1\rangle$). Following the postulate closely, we defined the `Den_expect` function in the execution model implementation:

```
(* Physics.v *)
Theorem Observables_and_States: forall rho observable Hbits,
Den_expect rho observable Hbits = Mtrace (Mmult rho observable Hbits).
Proof. reflexivity. Qed.
```

3.2.2 Measurement Probability

"Measurement Probability" postulate provides a specification to calculate the probabilitiy of each measurement output of the computational observables. Adhering to this postulate, we have defined the `Den_prob` function in the execution model implementation:

```
(* Physics.v *)
Theorem measurement_probability_postulate: forall rho projection Hbits,
Den_prob rho projection Hbits = Mtrace (Mmult rho projection Hbits).
Proof. reflexivity. Qed.
```

For the purpose of quantum computing, we specialize the `Den_prob` function with projection operators of computational basis:

```
(* Density.v *)
Definition Den_prob_0 (den: Matrix) (n t: nat) (Ht: t < n) (H: Mbits den = n): C.
  refine (Den_prob den (Qproj0_n_t n t Ht) _).
  simpl_bits.
  rewrite Qproj0_n_t_bits.
  apply H.
Defined.

Definition Den_prob_1 (den: Matrix) (n t: nat) (Ht: t < n) (H: Mbits den = n): C.
  refine (Den_prob den (Qproj1_n_t n t Ht) _).
  simpl_bits.
  rewrite Qproj1_n_t_bits.
  apply H.
Defined.
```

Here, `Den_prob_0` and `Den_prob_1` correspond to the probabilities of t-th qubit being 0 and 1, respectively. According to the postulate, `Qproj0_n_t n t Ht` and `Qproj0_n_t n t Ht` should be valid projection operators so that they satisfy the properties:
1) $PP = P$ 2) $P^* = P$ and 3) real non-negative trace (positive-definiteness):

```
(* Operator.v *)
Definition Projection (m: Matrix) := (forall H, Mmult m m H = m) /\ Qop_Hermitian m /\ Cge_0 (Mtrace m).
```

We have proven that projection operators for $|0\rangle$ and $|1\rangle$ are valid projection operators:

```
(* Operator.v *)
Lemma Qproj0_n_t_proj: forall (n t: nat) (Ht: _), Projection (Qproj0_n_t n t Ht).
Lemma Qproj1_n_t_proj: forall (n t: nat) (Ht: _), Projection (Qproj1_n_t n t Ht).
```

With these validations, it is safe to use `Den_prob_0` and `Den_prob_1`.

3.3 QASM Operations in the Execution Model

3.3.1 Projection Postulate (Measurement)

The **"Projection Postulate (Measurement)"** postulate informs us how quantum states are transformed during the measurement process. Strictly following the postulate, we defined `Den_measure` function in our execution model implementation:

```
(* Physics.v *)
Theorem projection_postulate: forall rho proj Hbits Hp Hm1 Hm2,
  Den_measure rho proj Hbits =
    Msmul
      (Cinv (Den_prob rho proj Hp))
      ( Mmult (
          Mmult proj rho Hm1
        ) proj Hm2).
Proof. reflexivity. Qed.
```

For quantum computing, we specialize the `Den_measure` function to measure in the computational basis:

```
(* Density.v *)
Definition Den_measure_0 (den: Matrix) (n t: nat) (Ht: t < n) (Hd: Mbits den = n): Matrix.
  refine (Den_measure den (Qproj0_n_t n t Ht) _).
  all: simpl_bits; simpl; lia.
Defined.
Definition Den_measure_1 (den: Matrix) (n t: nat) (Ht: t < n) (Hd: Mbits den = n): Matrix.
  refine (Den_measure den (Qproj1_n_t n t Ht) _).
  all: simpl_bits; simpl; lia.
Defined.
```

Here, `Qproj0_n_t n t Ht` and `Qproj0_n_t n t Ht` correspond to projections to the space where the t-th qubit is $|0\rangle$ and $|1\rangle$, respectively. We can now use `Den_measure_0` and `Den_measure_1` to execute a QASM2.0 measurement operation:

```
(* Program.v *)
Fixpoint Execute_measure_instr (qbit cbit: nat) (worlds: ManyWorld): ManyWorld.
Proof.
  destruct worlds as [|[qstate cstate prob nq Hq] t].
  - exact [].
  - destruct (lt_dec qbit nq).
    + specialize (Creal (Den_prob_0 qstate nq qbit l Hq)) as prob0.
      specialize (Creal (Den_prob_1 qstate nq qbit l Hq)) as prob1.
      destruct (Rgt_dec prob0 0), (Rgt_dec prob1 0).
      * refine ({|
          W_qstate := Den_measure_0 qstate nq qbit l Hq;
          W_cstate := tm_update cstate cbit false;
          W_prob := prob * prob0;
          W_num_qubits := nq;
        |} :: {|
          W_qstate := Den_measure_1 qstate nq qbit l Hq;
          W_cstate := tm_update cstate cbit true;
          W_prob := prob * prob1;
          W_num_qubits := nq;
        |} ::
        (Execute_measure_instr qbit cbit t)).
        apply Den_measure_0_bits.
        apply Den_measure_1_bits.
      * refine ({|
          W_qstate := Den_measure_0 qstate nq qbit l Hq;
          W_cstate := tm_update cstate cbit false;
          W_prob := prob * prob0;
          W_num_qubits := nq;
        |} ::
        (Execute_measure_instr qbit cbit t)).
        apply Den_measure_0_bits.
      * refine ({|
          W_qstate := Den_measure_1 qstate nq qbit l Hq;
          W_cstate := tm_update cstate cbit true;
          W_prob := prob * prob1;
          W_num_qubits := nq;
        |} ::
        (Execute_measure_instr qbit cbit t)).
        apply Den_measure_1_bits.
      * apply (Execute_measure_instr qbit cbit t). (* nop *)
  + apply (Execute_measure_instr qbit cbit t). (* nop *)
Defined.
```

Note that `Den_prob_0` and `Den_prob_0` are also used in `Execute_measure_instr`.

3.3.2 Time Evolution (Unitary Operation)

"Time Evolution" postulate describes how quantum states are changed during the time evolution, i.e., unitary operations such as single-qubit rotation or controlled-NOT (CNOT) gate in QASM2.0. Adhering strictly to this postulate, we defined `Den_unitary` function in the execution model implementation:

```
(* Physics.v *)
Theorem time_evoluation_postulate: forall rho uop H1 H2,
  Den_unitary rho uop H1 H2 = (Mmult (Mmult uop rho H1) (Mconjtrans uop) H2).
Proof. reflexivity. Qed.
```

As mentioned before, OpenQASM2.0 supports two built-in universal uniatry operations: rotation and CNOT. For each opertaion, we have defined a matrix that represents that operation respectively. It is necessary to show that the defined matrices are unitary.

1. Generalized Single-Qubit Rotation
   It is widely known that a single-qubit rotation for a single-qubit system is represented by:

$$\begin{pmatrix} e^{-i(\phi+\lambda)/2}\cos(\theta/2) & -e^{-i(\phi-\lambda)/2}\sin(\theta/2) \\ e^{i(\phi-\lambda)/2}\sin(\theta/2) & e^{i(\phi+\lambda)/2}\cos(\theta/2) \end{pmatrix}.$$

   Using the tensor product with the identity matrix, given a target qubit, we can build a single-qubit rotation matrix for multi-qubit system. The unitarity of these generalized single-qubit rotation can be proven using this lemma we previously proved:

```
(* Operator.v *)
Lemma Qop_unitary_TMprod: forall (m1 m2: Matrix),
  Qop_unitary m1 -> Qop_unitary m2 -> Qop_unitary (TMproduct m1 m2).

Lemma Qop_rot_unitary: forall (theta phi lambda: R), Qop_unitary (Qop_rot theta phi lambda).

Lemma Qop_sq_unitary: forall (n t: nat) (op: Matrix) (Ht: _) (Hop: _),
  Qop_unitary op -> Qop_unitary (Qop_sq n t op Ht Hop).
```

2. Generalized Controlled-NOT
   It is widely known that a CNOT gate that flips the second qubit if and only if the first qubit is one is represented by

$$\text{CNOT} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

   To build generalized CNOT gates that appliy to an arbitrary multi-qubit system with arbitrary control and target qubit, we used the tensor product with the identity matrix, as well as the SWAP operator:

$$\text{SWAP} := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

   Given control and target qubits, we can make a generalized multi-qubit CNOT gate by using an appropriate permutation of tensor product and SWAP matrix. In this case, it is significantly more complex to prove the unitarity, but it can be done using the previously mentioned `Qop_uniatry_TMProd` and a newly proved `Qop_uniatry_mult` lemma:

```
(* Operator.v *)
Lemma Qop_unitary_mult: forall (m1 m2: Matrix) (H: _),
  Qop_unitary m1 -> Qop_unitary m2 -> Qop_unitary (Mmult m1 m2 H).

Lemma Qop_cnot_unitary: forall (n qc qt: nat) (Hn: _) (Hc: _) (Ht: _),
  Qop_unitary (Qop_cnot n qc qt Hn Hc Ht).
```

We can now use `Den_unitary` execute a single-qubit rotation and CNOT operation in OpenQASM2.0:

```
(* Program.v *)
Fixpoint Execute_rotate_instr (theta phi lambda: R) (target: nat) (worlds: ManyWorld): ManyWorld.
Proof.
  destruct worlds as [|[qstate cstate prob nq Hq] t].
  - exact [].
  - destruct (lt_dec target nq).
    + refine ({|
        W_qstate := Den_unitary qstate (Qop_sq nq target (Qop_rot theta phi lambda) l _) _ _;
        W_cstate := cstate;
        W_prob := prob;
        W_num_qubits := nq;
      |} :: (Execute_rotate_instr theta phi lambda target t)).
      Unshelve.
```

```
      rewrite Den_unitary_bits.
      apply Hq.
      apply Qop_rot_bits.
      simpl_bits.
      rewrite Qop_sq_bits.
      lia.
      simpl_bits.
      reflexivity.
    + apply ({|
        W_qstate := qstate;
        W_cstate := cstate;
        W_prob := prob;
        W_num_qubits := nq;
        W_qstate_valid := Hq;
      |} :: (Execute_rotate_instr theta phi lambda target t)). (* nop *)
Defined.

Fixpoint Execute_cnot_instr (control target: nat) (worlds: ManyWorld): ManyWorld.
Proof.
  destruct worlds as [|[qstate cstate prob nq Hq] t].
  - exact [].
  - destruct (ge_dec nq 2), (lt_dec control nq), (lt_dec target nq).
    refine ({|
      W_qstate := Den_unitary qstate (Qop_cnot nq control target g l l0) _ _;
      W_cstate := cstate;
      W_prob := prob;
      W_num_qubits := nq;
      |} :: (Execute_cnot_instr control target t)).
    2-8: apply ({|
        W_qstate := qstate;
        W_cstate := cstate;
        W_prob := prob;
        W_num_qubits := nq;
        W_qstate_valid := Hq;
      |} :: (Execute_cnot_instr control target t)). (* nop *)
    Unshelve.
    rewrite Den_unitary_bits.
    apply Hq.
    simpl_bits.
    rewrite Qop_cnot_bits.
    lia.
    simpl_bits.
    reflexivity.
Defined.
```

3.4 Quantum Mixed States in the Execution Model

In this section, we aim to demonstrate that every `DensityMatrix` generated by our implementation fulfills three properties required by the first postulate. Note that these matrices have not yet bee proven to be a valid density matrices. In our implementation, `DensityMatrix` is defined by initial states and their transformations as set out in the postulates:

```
(* Density.v *)
Inductive DensityMatrix: nat -> Matrix -> Prop :=
| DensityMatrix_init (n: nat) (den: Matrix): InitialDensityMatrix n den -> DensityMatrix n den
| DensityMatrix_unitary (n: nat) (den uop: Matrix) (H1: _) (H2: _):
    DensityMatrix n den ->
    Qop_unitary uop ->
    DensityMatrix n (Den_unitary den uop H1 H2)
| DensityMatrix_measure (n: nat) (den proj: Matrix) (Hd: _):
    DensityMatrix n den ->
    Projection proj ->
    Den_prob den proj Hd <> 0 ->
    DensityMatrix n (Den_measure den proj Hd).
```

3.4.1 Self-Adjoint (Hermitian)

Given that the definition of self-adjoint operator is such that the complex conjugate is equivalent to itself ($\rho^* = \rho$), it is crucial to avoid multiplying non-real complex scalars to the self-adjoint operator

to maintain self-adjointness. In the projection(measurement) postulate, we multiply $\frac{1}{\mathrm{tr}(\rho P)}$ by a density matrix to calculate a quantum state post- measurement:

$$\rho = \begin{matrix}\text{state before}\\ \text{measurement}\end{matrix} \xrightarrow{\text{measurement}} \frac{P_\lambda \rho P_\lambda}{\mathrm{tr}\left(\rho P_\lambda\right)} = \frac{P_\lambda \rho P_\lambda}{\mathbf{P}_\rho\left(\lambda\right)} = \begin{matrix}\text{state after}\\ \text{measurement}\end{matrix}$$

Therefore, it becomes necessary to prove that $\mathrm{tr}\left(\rho P\right)$ (probability of P) is always real for any given projection operator P.
To prove $\mathrm{tr}\left(\rho P\right)$ is real, we have to establish that ρ is self-adjoint. Our goal, in proving reality of $\mathrm{tr}\left(\rho P\right)$, is to demonstrate that

$$\mathrm{tr}\left(\rho P\right)^* = \mathrm{tr}\left(\rho P\right)$$

in various base cases and inductive cases. By the way,

$$\mathrm{tr}\left(\rho P\right)^* = \mathrm{tr}\left(\left(\rho P\right)^*\right) = \mathrm{tr}\left(P^* \rho^*\right) = \mathrm{tr}\left(\rho^* P^*\right) = \mathrm{tr}\left(\rho^* P\right) \text{ ... (note that } P \text{ is self-adjoint)}$$

This transforms the problem of proving the reality of $\mathrm{tr}\left(\rho P\right)$ into one of proving self-adjointness. As a result, if we attempt to prove both properties simutaneously, we could establish both properties through induction:

```
(* Density.v *)
Lemma DensityMatrix_prob_real_Hermitian: forall (n: nat) (den: Matrix),
  DensityMatrix n den -> (forall proj, Projection proj ->
  (forall Hd, (Cimag (Den_prob den proj Hd) = 0%R))) /\ Qop_Hermitian den.
```

3.4.2 Positive Semi-definite

Proving positive semi-definiteness is more troublesome than the privious case. Given that semi-definiteness is defined as

$$\forall \left|\psi\right\rangle \in \mathbb{H}, \quad \left\langle\psi\right| A \left|\psi\right\rangle \geq 0\,,$$

it is crucial to avoid multiplying negative real scalars to positive semi-definite operator to maintain positive semi-definiteness. Like in the self-adjoint case, we have to multiply $\frac{1}{\mathrm{tr}(\rho P)}$ to a density matrix in order to calculate a post-measurement quantum state:

$$\rho = \begin{matrix}\text{state before}\\ \text{measurement}\end{matrix} \xrightarrow{\text{measurement}} \frac{P_\lambda \rho P_\lambda}{\mathrm{tr}\left(\rho P_\lambda\right)} = \frac{P_\lambda \rho P_\lambda}{\mathbf{P}_\rho\left(\lambda\right)} = \begin{matrix}\text{state after}\\ \text{measurement}\end{matrix}$$

So it is necessary to prove that $\mathrm{tr}\left(\rho P\right)$ (probability of P) is always non-negative for arbitrary projection operator P. Moreover, as the self-adjoint case, positive semi-definiteness is needed when proving non-negativity of $\mathrm{tr}\left(\rho P\right)$. Here's why.

$$\mathrm{tr}\left(\rho P\right) \geq 0 \xleftarrow{P=PP} \mathrm{tr}\left(\rho PP\right) \geq 0 \xleftarrow{\mathrm{tr}(AB)=\mathrm{tr}(BA)} \mathrm{tr}\left(P\rho P\right) \geq 0 \xleftarrow{\text{1st Lemma}} P\rho P \geq 0 \xleftarrow{\text{2nd Lemma}} \rho \geq 0$$

We have proven the corresponding "1st" and "2nd" lemmas:

```
(* Operator.v *)
(* 1st Lemma *)
Lemma Qop_positive_trace: forall (m: Matrix), Qop_positive m -> Cge_0 (Mtrace m).
(* 2nd Lemma *)
Lemma Qop_positive_mult_Hermitian: forall (m1 m2: Matrix) H1 H2,
  Qop_positive m1 -> Qop_Hermitian m2 -> Qop_positive (Mmult (Mmult m2 m1 H1) m2
  H2).
```

Consequently, the problem of proving non-negativity of $\mathrm{tr}\left(\rho P\right)$ has been transformed into one of proving positive semi-definiteness. Hence if both properties are attemptd to be proven simutaneously, we could establish both through induction:

```
(* Density.v *)
Lemma DensityMatrix_prob_pos_positive: forall (n: nat) (den: Matrix),
  DensityMatrix n den -> (forall proj, Projection proj ->
  (forall Hd, Cge_0 (Den_prob den proj Hd))) /\ Qop_positive den.
```

3.4.3 Trace 1

This case is relatively easier than the two cases before. Using induction, we could prove this lemma:

```
(* Density.v *)
Lemma DensityMatrix_normalized: forall (n: nat) (den: Matrix),
  DensityMatrix n den -> Den_normalized den.
```

3.4.4 Extension to Execution Level

Using the Lemmas we proved in the section 3.4.1 to 3.4.3, we can demonstrate that, for any given arbitrary QASM2.0 program, each quantum state of the world in the resulting program state (`ManyWorld`), produced by our execution model, adheres to the postulate about the quantum state (proposition 1):

```
Theorem all_states_Hermitian: forall program world,
  In world (Execute program) -> Qop_Hermitian (W_qstate world).

Theorem all_states_positive: forall program world,
  In world (Execute program) -> Qop_positive (W_qstate world).

Theorem all_states_trace_1: forall program world,
  In world (Execute program) -> Mtrace (W_qstate world) = 1.
```