$\mathcal{Q}$WIRE Practice: Formal Verification of Quantum Circuits in Coq

Robert Rand
rrand@seas.upenn.edu

Jennifer Paykin
jpaykin@seas.upenn.edu

Steve Zdancewic
stevez@cis.upenn.edu

University of Pennsylvania

We describe an embedding of the $\mathcal{Q}$WIRE quantum circuit language in the Coq proof assistant. This allows programmers to write quantum circuits using high-level abstractions and to prove properties of those circuits using Coq's theorem proving features. The implementation uses higher-order abstract syntax to represent variable binding and provides a type-checking algorithm for linear wire types, ensuring that quantum circuits are well-formed. We formalize a denotational semantics that interprets $\mathcal{Q}$WIRE circuits as superoperators on density matrices, and prove the correctness of some simple quantum programs.

1 Introduction

The last few years have witnessed the emergence of lightweight, scalable, and expressive quantum circuit languages such as Quipper [10] and LIQ$Ui|\rangle$ [22]. These languages adopt the QRAM model of quantum computation, in which a classical computer sends instructions to a quantum computer and receives back measurement results. Quipper and LIQ$Ui|\rangle$ programs classically produce circuits that can be executed on a quantum computer, simulated on a classical computer, or compiled using classical techniques to smaller, faster circuits. Since both languages are embedded inside general-purpose classical host languages (Haskell and F#), they can be used to build useful abstractions on top of quantum circuits, allowing for general purpose quantum programming.

As is the case with classical programs, however, quantum programs in these languages will invariably have bugs. Since quantum circuits are inherently expensive to run (either simulated or on a real quantum computer) and are difficult or impossible to debug at runtime, numerous techniques have been developed to verify properties of quantum programs.

The first step towards guaranteeing bug-free quantum programs is ensuring that every program corresponds to a valid quantum computation, meaning that a simulator or quantum computer running that program will not crash. In many cases this property can be enforced using type systems, as in the quantum lambda calculus, which uses linear types and guarantees type safety [20]. Along these lines, Proto-Quipper [19] adds linear types to a subset of Quipper, though this approach has not been extended to the full Quipper language.

Beyond simply ensuring quantum mechanical soundness, we might wish to statically analyze specific programs or families of programs and prove that their semantics matches a formal specification. Doing so requires a formal semantics for programs, such as unit vectors or density matrices. The specification can then be verified using model-checking [9], Hoare logic [23], proof assistants [3], or other techniques.

This work is supported in part by ONR MURI No. FA9550-16-1-0082 and NSF Grant No. CCF- 1421193.

Bob Coecke and Aleks Kissinger (Eds.):
14th International Conference on Quantum Physics and Logic (QPL)
EPTCS 266, 2018, pp. 119–132, doi:10.4204/EPTCS.266.8

Finally, one may wish to verify that two particular programs (or program fragments) have the same semantics. LIQ$Ui|\rangle$, in particular, has focused on efficient compilation of quantum circuits; similar projects have explored verified compilation passes in the case of reversible circuits [1].

The QWIRE programming language [14] is a small quantum circuit language embedded in a classical host language, which provides three core features: (1) a platform for high level quantum computing, with the expressiveness of embedded languages like Quipper [10] and LIQ$Ui|\rangle$[22]; (2) a linear type system that guarantees that generated circuits are well-formed and respect the laws of quantum mechanics; and (3) a concrete denotational semantics, specified in terms of density matrices, for proving properties and equivalences of quantum circuits.

In this paper we report on a ongoing effort to implement QWIRE in the Coq theorem prover [6] and formalize its denotational semantics, thereby providing a framework to formally verify the correctness of quantum programs and quantum program transformations.[1] We include two forms of the language, one using Coq variables and higher-order abstract syntax for ease of programming, and a simpler version without these features, for verification purposes, along with a simple translation from the former to the latter. We then use this bridge to prove properties about higher-order programs using QWIRE's denotational semantics.

The paper makes the following contributions:

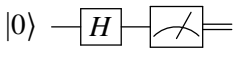
- We implement the QWIRE programming language in the Coq proof assistant, incorporating features such as dependently-typed circuits and proof-carrying code;
- We present a type checking algorithm for the linear type system of QWIRE using a representation of linear contexts based on the Linearity Monad [15];
- We formalize a denotational semantics for QWIRE circuits, interpreting them as superoperators on density matrices [13]; and
- We show how these semantics can be used to verify quantum programs, including a quantum coin flip, a protocol with *dynamic lifting*, and a simple unitary circuit.

Throughout the paper, we will assume some level of familiarity with both functional programming and proof assistants in the style of Coq or Agda. Readers unfamiliar with either of these concepts should consult the the introductory chapter of *Software Foundations* [17], which introduces the Coq language and theorem prover [6] used here.

2 Introduction to QWIRE programming

We start with some examples of QWIRE circuits implemented in Coq. The following circuit implements a quantum coin flip:[2]

```
Definition coin_flip : Box One Bit.
  box_ () ⇒
    gate_ x  ← init0 @();
    gate_ y  ← H @x;
    gate_ z  ← meas @y;
    output z.
```



[1] The Coq development for this paper is available at: https://github.com/jpaykin/QWIRE/tree/QPL2017.

[2] Unless otherwise indicated, all QWIRE circuit definitions end in an implicit Defined.

The type `Box One Bit` is the high-level interface to QWIRE circuits, representing a "boxed" circuit with no input wires (represented by the unit type `One`) and a `Bit`-valued output wire. We use Coq notations and tactics to construct the circuit. The `box_` tactic creates a boxed circuit from a function that takes an input pattern of wires and produces a circuit that uses those wires; `box_` also calls the linear type-checker on the result, to ensure that the circuit is well formed. In this example, the `coin_flip` circuit has no inputs, so the input pattern to the box is the empty pattern ().

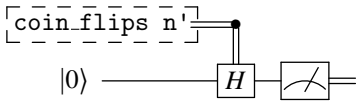
The body of the circuit is made up of a sequence of gate applications, where `init0`, `H`, and `meas` are all gates. These gates are applied to input patterns, whose types are determined by the gate being applied. In `coin_flip`, the pattern () has type `One`, `x` and `y` have type `Qubit`, and `z` has type `Bit`. The output of each gate is bound in the remainder of the circuit, and the circuit is terminated by an `output` statement.

QWIRE ensures that wires in a circuit are treated linearly, meaning that every wire is used exactly once as input and once as output. For example, the linear type system enforces the no-cloning property, rejecting the following bogus circuit.

```
Definition clone W : Box W (W ⊗ W).
  box_ w ⇒ output (w,w). (* Linear type checker cannot be satisfied. *) Abort.
```

QWIRE is compositional, meaning that a boxed circuit can be reused inside another circuit. Notice that only wires are treated linearly in QWIRE; a boxed circuit is a first-class Coq expression, and can be used as many times as necessary. Consider the following circuit, which flips a coin up to n times, returning 1 if and only if the coin always lands on heads, which occurs with probability $\frac{1}{2^n}$.

```
Fixpoint coin_flips (n : ℕ) : Box One Bit.
  box_ () ⇒ match n with    (* n:ℕ is either 0 or the successor of some n' *)
             | 0    ⇒ gate_ x ← new1 @(); output x
             | S n' ⇒ let_  c     ← unbox (coin_flips n') ();
                      gate_ q     ← init0 @();
                      gate_ (c,q) ← bit_ctrl H @(c,q);
                      gate_ ()    ← discard @c;
                      gate_ b     ← meas @q;
                      output b
             end.
```



The `unbox` operator feeds the input pattern () into the boxed circuit `coin_flips n'`, and the `let_` operation binds the output of the recursive call in the rest of the circuit. The `let_` and `unbox` operations are separate constructs; for example, `box_ w ⇒ unbox b w` is just the η-expansion of a boxed circuit `b`.

QWIRE also allows for *dynamic lifting* [14], in which measurement results from the quantum circuit are dynamically processed as classical data inside of Coq. This `lift` operation measures a qubit (or any collection of wires) and produces a boolean (or a tuple of booleans). We use the result of the measurement to decide which gates to apply in the remainder of the circuit. The following variation on `coin_flips` uses dynamic lifting and calls the unary `coin_flip` circuit from the start of this section.

```
Fixpoint coin_flips' (n : ℕ) : Box One Bit.
  box_ () ⇒ match n with
             | 0    ⇒ gate_ q ← new1 @(); output q
             | S n' ⇒ let_  q ← unbox (coin_flips' n') ();
                      lift_ x ← q;
                      if x then unbox coin_flip ()
                           else gate_ q ← new0 @(); output q
             end.
```

Further examples can be found in the online development, as well as in the original QWIRE paper [14].

3 Implementing QWIRE in Coq

At its core, a QWIRE circuit is a sequence of gates applied to wires. Each wire is described by a *wire type* `W`, which is either the unit type (has no data), a bit or qubit, or a tuple of wire types. In Coq we represent wire types as an inductively defined data type `WType` as follows:

```
Inductive WType := One | Bit | Qubit | Tensor : WType → WType → WType.
```

We use the Coq notation ``$W_1 \otimes W_2$" for `Tensor` W_1 W_2.

Gates are indexed by a pair of wire types—a gate of type `Gate` W_1 W_2 takes an input wire of type W_1 and outputs a wire of type W_2. In our setting, gates will include a universal set of unitary gates, as well as initialization, measurement, and control.[3]

```
Inductive Unitary : WType → Set := (* other unitary gates omitted *)
  | H         : Unitary Qubit (* Hadamard gate *)
  | control   : ∀ {W} (U : Unitary W), Unitary (Qubit ⊗ W)
  | transpose : ∀ {W} (U : Unitary W), Unitary W.
Inductive Gate : WType → WType → Set :=
  | U    : ∀ {W} (u : Unitary W), Gate W W
  | init : Gate One Qubit
  | meas : Gate Qubit Bit
  | discard : Gate Bit One.
```

The curly braces surrounding the type argument `W` indicate that `W` is an implicit argument, meaning that it can be automatically inferred from the other arguments. We further define `U` to be a *coercion* from unitaries to gates, meaning that for `u : Unitary W`, we can simply write `u` for `U u : Gate W W`.

An open circuit (one with free input wires) has Coq type `Circuit` Γ `W`, where Γ is a typing context of input wires and W is an output wire type. As an example, the circuit

```
gate_ w1' ← H @w1; gate_ w2' ← meas @w2; output (w2',w1')
```

has type `Circuit (w1:Qubit, w2:Qubit) (Bit ⊗ Qubit)`.

A typing context of type `Ctx` is a partial map from variables (represented concretely as natural numbers) to wire types, which is represented as `list (option WType)`. In this representation, the variable i is mapped to W if the ith element in the list is `Some` W, and is undefined if the ith element is `None`.

The *disjoint merge* operation ⩔ ensures that the same wire cannot be used in two separate parts of a circuit. Mathematically, it is defined on two typing contexts as follows:

```
[]   ⩔  Γ₂                      = Γ₂
Γ₁   ⩔  []                      = Γ₁
None   :: Γ₁  ⩔  None   :: Γ₂ = None   :: (Γ₁ ⩔ Γ₂)
Some W :: Γ₁  ⩔  None   :: Γ₂ = Some W :: (Γ₁ ⩔ Γ₂)
None   :: Γ₁  ⩔  Some W :: Γ₂ = Some W :: (Γ₁ ⩔ Γ₂)
```

Since disjoint merge is a partial function we represent it in Coq as a relation on possibly invalid contexts, `OCtx = Invalid | Valid Ctx`. For convenience, most operations on contexts are lifted to work with `OCtx` values, and so the type signature of the merge operation ⩔ is `OCtx → OCtx → OCtx`.

Wires in a context Γ can be collected into a pattern `Pat` Γ `W` to construct the wire type `W`. A pattern is just a tuple of wires of base types, meaning that all variables in a pattern have type `Bit` or `Qubit`. We use Coq's dependent types to express logical predicates that constrain how patterns can be constructed.

[3] The set of gates need not be fixed; Rennela and Staton [18] explore how to use gates to extend QWIRE with recursive types.

```
Inductive Circuit' : OCtx → WType → Set :=
| output' : ∀ {Γ W}, Pat Γ W → Circuit' Γ W
| gate'   : ∀ {Γ Γ₁ Γ₂ W₁ W₂ W},
            is_valid (Γ₁ ⩔ Γ) → is_valid (Γ₂ ⩔ Γ) → Gate W₁ W₂ →
            Pat Γ₁ W₁ → Pat Γ₂ W₂ → Circuit' (Γ₂ ⩔ Γ) W → Circuit' (Γ₁ ⩔ Γ) W
| lift'   : ∀ {Γ₁ Γ₂ W W'}, is_valid (Γ₁ ⩔ Γ₂) →
            Pat Γ₁ W → (interpret W → Circuit' Γ₂ W') → Circuit' (Γ₁ ⩔ Γ₂) W'.
```

Figure 1: Definition of $\mathcal{Q}$WIRE circuits using an explicit representation of variable binding. We call this type `Circuit'`, reserving the name `Circuit` for the higher-order abstract syntax representation (Figure 2).

```
Inductive Pat : OCtx → WType → Set :=
| unit  : Pat (Valid []) One
| qubit : ∀ (x : ℕ) (Γ : Ctx), SingletonCtx x Qubit Γ → Pat (Valid Γ) Qubit
| bit   : ∀ (x : ℕ) (Γ : Ctx), SingletonCtx x Bit   Γ → Pat (Valid Γ) Bit
| pair  : ∀ Γ₁ Γ₂ W₁ W₂, is_valid(Γ₁ ⩔ Γ₂)→Pat Γ₁ W₁→Pat Γ₂ W₂→Pat (Γ₁ ⩔ Γ₂) (W₁⊗W₂).
```

The `pair` constructor (for which we use notation (p1,p2)) ensures that the wires in `p1` are disjoint from those in `p2` by calling out to the `is_valid` predicate, which checks whether the result of the merge is well-defined. The `qubit` and `bit` patterns are variable constructors that are only valid in contexts that contain the exact variable being introduced. The predicate `SingletonCtx x W Γ` ensures that the context Γ contains only the single wire x of type W.

Circuits Like patterns, circuits are indexed by an input context and an output type. There are only three syntactic forms for circuits: *output*, *gate application*, and *dynamic lifting*. Figure 1 defines circuits as an inductive data type indexed by the input typing context and the output wire type.

An output circuit `output p` is just a pattern. A gate application, which we write with syntactic sugar as `gate_ p2 ← g @p1; C`, is made up of a gate `g : Gate W₁ W₂`, an input pattern `p1 : Pat Γ₁ W₁`, an output pattern `p2 : Pat Γ₂ W₂`, and a circuit `C : Circuit' (Γ₁ ⩔ Γ₂) W'`. The intended meaning is that `p1` is the input to the gate `g`, and its output is bound to `p2` in the continuation `C`. Thus the variables `Γ₂` that made up `p2` are also free in C, whereas the variables `Γ₁` in `p1` are no longer available in C, enforcing linearity. The `is_valid` predicates enforce that all of the context arguments to circuits must be well-defined.

The lift operation, which we write `lift_ x ← p; C` takes as input a pattern `p : Pat Γ₁ W` and a function `fun x ⇒ C` that takes the classical interpretation of that data and produces another circuit. The intended semantics is that the circuit will measure the wires `p` (if they are not already bit-valued) and pass the result as ordinary Coq data to the function. In particular, both bits and qubits will result in boolean values being provided to the continuation, and tensors will be interpreted as pairs.

A boxed circuit, written `box_ p ⇒ C` is a pair of a pattern and a circuit.

```
Inductive Box' : WType → WType → Set :=
| box' : ∀ {W₁ W₂ Γ}, Pat Γ W₁ → Circuit' Γ W₂ → Box' W₁ W₂.
```

Composition and Higher-Order Abstract Syntax The minimal embedding of $\mathcal{Q}$WIRE in Figure 1 lacks a number of features of the language, including composition and unboxing. The `unbox` operator takes in a boxed circuit and an input pattern and produces a circuit. It has the following signature:

```
Definition unbox {Γ W₁ W₂} (b : Box' W₁ W₂) (p : Pat Γ W₁) : Circuit' Γ W₂.
```

```
Inductive Circuit : OCtx → WType → Set :=
| output : ∀ {Γ Γ' w}, (Γ = Γ') → Pat Γ w → Circuit Γ' w
| gate   : ∀ {Γ Γ₁ Γ₁' w1 w2 w}, is_valid Γ₁' → (Γ₁' = Γ₁ ⩒ Γ)
        → Gate w1 w2 → Pat Γ₁ w1
        → (∀ {Γ₂ Γ₂'}, is_valid Γ₂' → (Γ₂' = Γ₂ ⩒ Γ) → Pat Γ₂ w2 → Circuit Γ₂' w)
        → Circuit Γ₁' w
| lift : ∀ {Γ₁ Γ₂ Γ w w'}, is_valid Γ → (Γ = Γ₁ ⩒ Γ₂)
        → Pat Γ₁ w → (interpret w → Circuit Γ₂ w')
        → Circuit Γ w'.
```

Figure 2: A definition of QWIRE circuits using higher-order abstract syntax.

The intended β-reduction rule should have the form `unbox (box_ p ⇒ C) p' = C[p'/p]` where `C[p'/p]` is a substitution of the variables in `p'` for those in `p` in the circuit `C`.

The traditional solution to this problem involves defining a number of substitution functions and proving appropriate typing relations for each operation. Although this approach is viable, it is often tedious and introduces a large amount of complexity, especially in linear systems. An alternative approach is the technique of higher-order abstract syntax (HOAS) [16], in which variable bindings in an embedded language are represented as functions in the host language. This means that the language designer does not have to define substitution functions and prove their correctness, and that variables in the embedded language have the same weight as variables in the host language.

The HOAS approach to boxed circuits treats a box as a function from patterns to circuits:

```
Inductive Box : WType → WType → Set :=
| box : (∀ {Γ}, Pat Γ W₁ → Circuit Γ W₂) → Box W₁ W₂.
```

The `unbox` operation then simply destructs the box and applies the function appropriately.

```
Definition unbox {Γ W₁ W₂} (b : Box W₁ W₂) (p : Pat Γ W₁) : Circuit Γ W₂ :=
  match b with box f ⇒ f p end.
```

We can take a similar approach for the binding pattern in gate application, as shown in Figure 2. In this setting, output of a gate application is represented by a function from the output pattern to a new circuit. The other difference between the HOAS presentation of circuits and the "flat" representation `Circuit'` in the previous section is that in addition to proofs of validity about merged contexts, we introduce fresh arguments for the output context of each circuit. This is due to a technical limitation of Coq pattern matching, and the result makes it possible for us to write the examples in Section 2.

With this machinery in place, we can define composition as a meta-operation on circuits. For conciseness, we give its definition as a sequence of β-reduction rules, omitting the proof arguments.

```
Fixpoint compose {Γ₁ Γ₁' W Γ W'}  (c : Circuit Γ₁ W)
         (f : ∀ {Γ₂ Γ₂'}, (Γ₂' = Γ₂ ⩒ Γ) → is_valid Γ₂' → Pat Γ₂ W → Circuit Γ₂' W')
         : is_valid Γ₁' → (Γ₁' = Γ₁ ⩒ Γ) → Circuit Γ₁' W'.

    compose (output p)    f = f p
    compose (gate g p1 h) f = gate g p1 (fun p2 ⇒ compose (h p2) f)
    compose (lift p h)    f = lift p (fun x ⇒ compose (h x) f)
```

The QWIRE type checker In type-checking QWIRE circuits, we are asked to solve equations of the form $\Gamma_1 \uplus \cdots \uplus \Gamma_n = \Gamma_1' \uplus \cdots \uplus \Gamma_m'$ and `is_valid` $(\Gamma_1 \uplus \cdots \uplus \Gamma_n)$, in order to enforce the linearity of wires.

The first of these goals can be discharged with a automated proof tactic for solving systems of commutative monoids. In the higher-order abstract syntax version of circuits, these predicates may also contain *evars*, existentially quantified Coq variables. By starting at the leaves of the typing derivation, we can ensure that every equation of the form $\Gamma_1 \cup \cdots \cup \Gamma_n = \Gamma_1' \cup \cdots \cup \Gamma_m'$ has at most one evar. We can then cancel out all variables that appear on both sides of the equation, and unify the evar with what remains on the opposite side.

Once repeated applications of `monoid` have replaced all existential variables with regular Coq variables, we can prove goals of the form `is_valid` $(\Gamma_1 \cup \cdots \cup \Gamma_n)$. We take advantage of the fact that a set of finite contexts is disjoint if and only if the contexts are all pairwise disjoint. Our custom tactic for solving these goals extracts all proofs of pairwise disjointness from hypotheses and then reduces the goal to a conjunction of pairwise disjointness claims `is_valid`$(\Gamma_i \cup \Gamma_j)$. It then applies proofs of these claim, where available, from the hypotheses.

4 Denotational Semantics

4.1 The Matrix Library

The denotational semantics of QWIRE is implemented using a matrix library created specifically for this purpose. Matrices are simply functions from pairs of natural numbers to complex numbers.[4]

```
Definition Matrix (m n : ℕ) := ℕ → ℕ → ℂ.
```

The arguments *m* and *n*, which are the dimensions of the matrix, are not used directly in the definition, but they are useful to define certain operations on matrices, such as the Kronecker product and matrix multiplication, which depend on these dimensions. They are also useful as an informal annotation that aids the programmer. We say a matrix is well-formed when it is zero-valued outside of its domain.

```
Definition WF_Matrix {m n} (M : Matrix m n) : ℙ := ∀ i j, i ≥ m ∨ i ≥ n → M i j = 0.
```

This library is designed to facilitate reasoning about and computing with matrices. Treating matrices as functions allows us to easily express otherwise complicated matrix operations. Consider the definitions of Kronecker product (⊗) and complex conjugate transpose (†), where `Cconj` is the complex conjugate:

```
Definition kron {m n o p} (A : Matrix m n) (B : Matrix o p) : Matrix (m*n) (o*p) :=
  fun x y ⇒ A (x / o) (y / p) * B (x mod o) (y mod p).
Definition ctrans {m n} (A : Matrix m n) : Matrix n m := fun x y ⇒ Cconj (A y x).
```

We represent complex numbers using an adaptation of Coquelicot's `Complex` library [4]. This representation has the advantage of being a straightforward extension of the Coq real numbers, allowing us to easily extend the Coq Standard Library's powerful Linear Real Arithmetic solver (`lra`) to both complex numbers and matrices. The tactics we define (termed `clra` for complex numbers and `mlra` for matrices) allow us to trivially prove that complex conjugate and complex conjugate transpose are involutive:

```
Lemma conj_involutive : ∀ (c : ℂ), Cconj (Cconj c) = c.  Proof. intros. clra. Qed.
Lemma ctrans_involutive : ∀ {m n} (A : Matrix m n), A†† = A. Proof. intros. mlra. Qed.
```

[4] As a Coq technicality, note that matrices are only equal up to functional extensionality.

4.2 Density Matrices

QWIRE programs are interpreted as superoperators over density matrices, following the denotational semantics described by Paykin *et al.* [14]. We use a density matrix representation over the standard unit vector representation (used for example by Boender *et al.* [3]) because density matrices represent probability distributions (introduced via measurement) over quantum states directly, as opposed to the unit vector representation which must be embedded inside a probability monad.

We start with some preliminary definitions. A unitary matrix is a well-formed square matrix A such that $A^\dagger \times A$ is the identity.

```
Definition is_unitary {n} (A : Matrix n n) := WF_Matrix A  ∧  A† × A = Id n.
```

A pure state of a quantum system is one that corresponds to a unit vector $|\phi\rangle$. An equivalent representation is that of square matrices ρ such that $\rho \times \rho = \rho$.

```
Definition Pure_State {n} (ρ : Matrix n n) : ℙ := WF_Matrix ρ  ∧  ρ = ρ × ρ.
```

A density matrix, or mixed state, is a linear combination of pure states representing the probability of each pure state.

```
Inductive  Mixed_State {n} (ρ : Matrix n n) : ℙ :=
| Pure_S : ∀ ρ, Pure_State ρ → Mixed_State ρ
| Mix_S  : ∀ (p : ℝ) ρ1 ρ2, 0 < p < 1
        → Mixed_State ρ1 → Mixed_State ρ2 → Mixed_State (p .* ρ1 .+ (1-p) .* ρ2).
```

Note that every mixed state is also well-formed, since scaling and addition preserve well-formedness.

A superoperator is a function on square matrices that takes mixed states to mixed states.

```
Definition Superoperator m n := Matrix m m → Matrix n n.
Definition WF_Superoperator m n (f : Superoperator m n) :=
    ∀ (ρ : Matrix m m), Mixed_State ρ → Mixed_State (f ρ).
```

Any $m \times n$ matrix can be lifted to a superoperator from n to m as follows:

```
Definition super {m n} (A : Matrix m n) : Superoperator n m := fun ρ ⇒ A × ρ × A†.
```

4.3 Denotation of QWIRE

Types, Contexts and Gates In order to interpret circuits as superoperators over density matrices, we will also give types, contexts, gates, and patterns interpretations in linear algebra. For clarity we write $[\![-]\!]$ for the denotation of a variety of QWIRE objects, which we express via a Coq type class.

```
Class Denote source target := { denote : source → target }.
Notation "⟦ x ⟧" := (denote x) (at level 10).
```

We interpret every wire type as the number of primitive (`Bit` or `Qubit`) wires in that type, so $[\![\texttt{Qubit} \otimes (\texttt{One} \otimes \texttt{Bit})]\!] = 2$. Contexts and `OCtx`s are similarly denoted by the number of wires they contain.

Every gate of type `Unitary W` corresponds to a unitary matrix of dimension $2^{[\![W]\!]} \times 2^{[\![W]\!]}$. We omit the implemented gate set and their corresponding matrices here, but in the development we prove that every denoted unitary satisfies the `is_unitary` predicate defined above.

```
Lemma unitary_gate_unitary : ∀ {W} (u : Unitary W), is_unitary ⟦u⟧.
```

A gate of type `Gate W1 W2` corresponds to a superoperator as follows:

```
Definition denote_gate {W1 W2} (g : Gate W1 W2) : Superoperator 2^⟦W1⟧ 2^⟦W2⟧ :=
  match g with
  | U u          ⇒ super ⟦u⟧
  | init0 | new0 ⇒ super |0⟩
  | init1 | new1 ⇒ super |1⟩
  | meas         ⇒ fun ρ ⇒ super |0⟩⟨0| ρ .+ super |1⟩⟨1| ρ
  | discard      ⇒ fun ρ ⇒ super ⟨0| ρ .+ super ⟨1| ρ
  end.
Instance Denote_Gate {W1 W2} : Denote (Gate W1 W2) (Superoperator 2^⟦W1⟧ 2^⟦W2⟧) :=
   {| denote := denote_gate |}.
```

When applying a gate to a subset of a quantum system, however, we will need a generalization of the `denote_gate` operation that applies the gate to the first part of a quantum system.

```
Definition denote_gate' n {W1 W2} (g : Gate W1 W2) : Superoperator 2^⟦W1⟧*2^n 2^⟦W2⟧*2^n.
```

What we previously wrote as `denote_gate` is simply `denote_gate' 0`.

Patterns and Flat Circuits Patterns of type `Pat Γ W` are interpreted as permutation matrices of dimension $2^{⟦\Gamma⟧} \times 2^{⟦W⟧}$. These matrices are constructed via multiple applications of a `swap` matrix, as follows: In general we will want to swap the positions of two arbitrary qubits in a system; to swap qubit 0 with qubit 2 in a 3-qubit system, we invoke `swap2 0 2` $= (I_2 \otimes \texttt{swap})(\texttt{swap} \otimes I_2)(I_2 \otimes \texttt{swap})$.

$$\texttt{swap} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

First, every pattern is interpreted as a list of indices indicating the wire number (counting from 0) that each variable refers to. So the pattern `(3,0) : Pat [Some Qubit, None, None, Some Bit] (Bit ⊗ Qubit)` corresponds to the list $[1;0]$, because variable 3 corresponds the wire numbered 1 and variable 0 corresponds to wire 0 in the circuit. This list `ls` is then turned into an association list $[(0,1);(1,0)]$, mapping variable `i` to `ls[i]`. These pairs are then interpreted as a series of calls to `swap2`.

As for gates, the function `denote_pat_in` follows a similar algorithm, but allows us to interpret a pattern `p` inside a larger context of variables. Its signature is as follows:

```
Definition denote_pat_in Γ' {Γ W} (p : Pat Γ W): Matrix 2^⟦Γ ⩒ Γ'⟧ (2^⟦W⟧ * 2^⟦Γ'⟧).
Instance Denote_Pat {Γ W} : Denote (Pat Γ W) (Matrix 2^⟦Γ⟧ 2^⟦W⟧) :=
    {| denote := denote_pat_in (Valid []) |}.
```

Finally, we interpret circuits as superoperators on density matrices. To do this, we start with the "flat" representation `Circuit'` that does not use higher-order abstract syntax. The higher-order abstract syntax representation is useful for defining the meta-level operations on circuits, but the "flat" circuits from Figure 1 use concrete variables, thus making them a useful intermediate representation for the semantics.

Consider an output circuit `output' p`, where `p : Pat Γ W`. The interpretation of this circuit is the superoperator obtained from the denotation of `p`: ⟦`output' p`⟧ = `super` ⟦`p`⟧.

Next, consider `gate_ p2 ← g @p1; C`, where `g : Gate W1 W2`, $\texttt{p}_\texttt{i}$ `: Pat` $\Gamma_\texttt{i}$ $\texttt{W}_\texttt{i}$ and `C : Circuit'` $(\Gamma_2 \Cup \Gamma)$ `W`. The interpretation of `C` has type `Superoperator` $2^{⟦\Gamma_2 \Cup \Gamma⟧}\ 2^{⟦W⟧}$, so we need to compose ⟦`C`⟧ with a superoperator from $2^{⟦\Gamma_1⟧*2^{⟦\Gamma⟧}}$ to $2^{⟦\Gamma_2⟧*2^{⟦\Gamma⟧}}$ which we obtain by composing `denote_gate'` with the results of `denote_pat_in`, to rearrange the quantum system appropriately:

```
⟦gate' g p1 p2 C'⟧ = ⟦C'⟧ ∘ super (⟦p2⟧† ⊗ Id 2^⟦Γ⟧)
                          ∘ denote_gate' ⟦Γ⟧ g ∘ super (⟦p1⟧ ⊗ Id 2^⟦Γ⟧)
```

Finally, consider a lift circuit, `lift' p f`, where `p : Pat Γ1 W` and `f : interpret W → Circuit' Γ2 W'`. When `W` is a qubit, `interpret W = bool`, and the lift operation would measure the qubit and sum over the results of ⟦`f true`⟧ and ⟦`f false`⟧. More generally, consider

```
Definition f' : interpret W → Superoperator 2^⟦Γ1 ∪ Γ2⟧ 2^⟦Γ2⟧
    := fun x ⇒ ⟦f x⟧ ∘ (super ((kets x)† ⊗ Id 2^⟦Γ2⟧)) ∘ (super (denote_pat_in Γ2 p))
```

Here, `kets {W} : interpret W → Matrix 2^⟦W⟧ 1` is the basis representation of the input value of type `interpret W`. By transposing `kets x` and expanding it via the `super` operation, we pick out the partial density matrix corresponding to that measurement branch. Next, since all wire types are finite, we can enumerate all values of type `interpret W` in a list via the operation `get_interpretations W`. By mapping `f'` over this list, we obtain each of the actual measurement branches as superoperators. Now we can simply perform pointwise addition of the superoperators, and compose with the pattern `p` to organize the wires in order:

```
⟦lift' p f⟧ = fold_left Splus (map f' (get_interpretations W)) SZero
```

A flat box is also interpreted as a superoperator: ⟦`box' p C`⟧ = ⟦`C`⟧ ∘ `super` ⟦`p`⟧†

HOAS Circuits To denote the HOAS version of circuits, we first map them to our representation of "flat" circuits, which involves instantiating the output patterns of HOAS gates with a particular concrete pattern. We do this via an operation `fresh_pat` that takes as input a context and a type, and produces a pattern of that type whose domain (`fresh_pat_ctx`) is disjoint from the input context.

```
Definition fresh_pat (Γ : OCtx) (W : WType) : Pat (fresh_pat_ctx Γ W) W.
```

Using `fresh_pat` we can define a function that converts HOAS circuits and boxes into flat circuits and boxes. We leave off implicit proof and `OCtx` arguments for legibility.

```
Program Fixpoint from_HOAS {Γ W} (c : Circuit Γ W) : Circuit' Γ W :=
 match c with
 | output p    ⇒ output' p
 | gate g p1 f ⇒ gate' g p1 p2 (from_HOAS (f (fresh_pat _ _)))
 | lift p f    ⇒ lift' p (fun x ⇒ from_HOAS (f x))
 end.
Program Definition from_HOAS_Box {W1 W2} (b : Box W1 W2) : Flat_Box W1 W2 :=
  match b with box f ⇒ let p := fresh_pat [] W1 in box' p (from_HOAS (f p)) end.
```

The denotation of a HOAS circuit is exactly the denotation of its corresponding flat circuit.

5 A Taste of Verification

When a circuit is closed, that is when it has no input, it represents a preparation of a quantum state. In many cases, a programmer may know what state their program should prepare, and our verification framework allows them to compare the denotation of the circuit with the desired density matrix directly.

Consider, for instance, the coin flip circuit in Section 2. In the online development we prove that the denotation of `coin_flip` corresponds to the matrix `even_toss`$= \begin{pmatrix} 1/2 & 0 \\ 0 & 1/2 \end{pmatrix}$ as follows:

```
Lemma flip_toss : ⟦ coin_flip ⟧ (Id 1)  = even_toss.
```

The Coq proof script used to prove this lemma is only twelve lines long, and calls out to a number of specialized tactics, including `Msimpl`, `Csimpl` and `Csolve`, designed to simplify and prove equality between matrices and complex numbers.

The combination of Coq and QWIRE truly shines in its ability to prove more complex properties and prove properties of *families* of circuits, not merely circuits themselves. For example, consider the Deutsch-Jozsa problem, where we want to verify a property of a family of circuits with any number of input qubits, that holds for every appropriate unitary matrix U_f. This sort of property requires induction over the number of inputs and the full power of a proof assistant.

Open circuits, which correspond to arbitrary superoperators, are even more interesting from the perspective of verification. Consider the following circuit which composes a unitary gate with its transpose:

```
Definition unitary_trans {W} (U : Unitary W) : Box W W.
  box_ p ⇒
    gate_ p ← U @p;
    gate_ p ← transpose U @p;
    output p.
```


```
Lemma unitary_trans_id : ∀ W (U : Unitary W) ρ,
    WF_Matrix (2^⟦W⟧) (2^⟦W⟧) ρ → ⟦unitary_trans U⟧ρ = ⟦id_circ W⟧ρ.
```

This correctness of this circuit holds for any unitary gate regardless of its input size. The proof of this lemma takes fewer than 20 lines of code, and proceeds using common facts from linear algebra, such as the fact that `Id n × A = A`, along with fact that the denotation of `U` is in fact unitary, that is, ⟦U⟧† ⟦U⟧ = `Id`. We also rely on the fact that the denotation of every pattern in this circuit is actually an identity matrix.

Another useful sanity check ensures that the denotation of a `lift` circuit is equivalent to applying a measurement gate. For example, consider `lift_meas`, which measures a qubit via `lift` and then reconstructs a bit-valued wire from the result of the measurement. The resulting circuit is equivalent to the circuit that simply applies a measurement gate.

```
Definition lift_meas : Box Qubit Bit.
  box_ q ⇒
    lift_ x ← q;
    gate_ p ← (if x then new1 else new0) @();
    output p.
Lemma lift_meas_correct : ∀ ρ, WF_Matrix ρ
   → ⟦lift_meas⟧ ρ = ⟦boxed_gate meas⟧ ρ.
```

6 Related and Future Work

The Coq development described in this paper is very much still a work in progress. There are a small number of lemmas in the underlying matrix library that we have not yet formally proved, although they are known facts about linear algebra. In addition, we have not yet formally proved that the denotation of every circuit is a well-formed superoperator over density matrices, or that the denotation of the composition of two circuits is equal to the composition of their denotations.

After completing this work, there are a number of exciting areas to explore in the future.

Verified Compilation By verifying the equivalence of circuit transformations like `unitary_trans_id` from Section 5, we see QWIRE as a prime language in which to compile quantum programs. The area of verified compilers has gained a lot of traction in recent years, inspired by the success of the CompCert C compiler [12]. Towards this direction, Amy *et al.* [1] developed a verified, lightly-optimizing compiler for reversible circuits, which can encompass unitary quantum circuits.

We can also provide insight into a relationship between QWIRE circuits and QASM [7], a quantum assembly-like language that has gained widespread use in quantum simulators and IBM's Quantum Experience [11]. The largest difference between QWIRE and QASM is our use of variable binding, using abstract variables such as x, y, and z and allowing qubits to be renamed as in `let_ y ← output x; C'`. In

comparison, QASM operates on a concrete set of quantum registers (*e.g.,* qubits 1, 2, or 3) that cannot be renamed. In the development we provide a version of QWIRE that similarly operates directly on named qubits. Our denotational semantics extends directly to this representation, and we can compile from QWIRE to these "assembly-level" circuits. Future work could formally establish a relationship between the assembly-level version of QWIRE and QASM, allowing us to prove properties of QASM programs, and prove that compilation is sound with respect to the denotational semantics.

A More Efficient Backend Unfortunately, we have so far struggled to prove properties of (non-parametric) circuits with more than a few qubits, indicating that scalability will be a challenge moving forward. A key contributor to the scalability of our theorems is the underlying matrix library, which is not currently optimized in any significant way. Of course, any quantum simulator will be intractable on large enough circuits, as density matrices are always exponential in the size of the corresponding circuit.

In the near future, we will be transitioning the linear algebra back-end of our development to one of several existing projects, in the hopes of making it more efficient. Boender *et al.* [3] present one candidate, a library developed reasoning about the correctness of quantum protocols using pure states. Another candidate is the Coq Effective Algebra Library (CoqEAL) [5], which is designed specifically to allow matrix computation inside Coq. This library allows easy translation between it matrices and those of the Mathematical Components library [2], which in turn was designed with an eye towards verification. The two libraries together may substantially increase our ability to run and reason about QWIRE programs. We can also simulate QWIRE programs by extraction to OCaml.

Theory of QWIRE In this paper we focus on the denotational semantics of QWIRE, but many other aspects of the meta-theory are left to be explored. Rennela and Staton [18] present a categorical model of EWire, a close variant of QWIRE, as an enriched category. Their model also allows for additional features such as quantum data types in the style of Quipper [10].

The equational theory of quantum circuits is another area left for future work. For example, Staton [21] presents an axiomatization of the relationship between measurement, qubit initialization, and a limited set of unitary gates. In the future we hope to adapt Staton's work to QWIRE and thereby reason syntactically, rather than semantically, about the equivalence of QWIRE circuits. An equational theory is also key to integrating QWIRE with dependent types [14].

Verifying Higher-Order Programs The use of dependent types was a driving factor for both the development of QWIRE and the choice to embed it in Coq specifically. While dependent types power Coq's verification capabilities, they're also key to our representation of circuits. For example, in the development we implement a dependently-typed version of the Quantum Fourier Transform, as described in the introduction to QWIRE [14]. This paper hasn't focused on dependent types, or high level programming in QWIRE generally, but both will feature heavily in future QWIRE development.

Greater use of the host language and classical programming abstractions will also allow to further push the boundaries of quantum verification. While current QWIRE programs can be thought of as "simply circuits," such direct proofs will become increasingly difficult as we add on layers of abstraction—just as it would be nearly impossible to prove interesting program properties by reasoning about the underlying classical circuits. This will require new approaches to quantum verification, from quantum weakest precondition reasoning [8] (expanded into a Hoare-like logic in [23, 24]), to the forms of equational reasoning described above.

References

[1] Matthew Amy, Martin Roetteler & Krysta M Svore (2017): *Verified compilation of space-efficient reversible circuits*. In: *International Conference on Computer Aided Verification*, Springer, pp. 3–21, doi:10.1007/978-3-319-63390-9_1.

[2] Assia Mahboubi and Enrico Tassi (2016): *Mathematical Components*. Electronic resource, available from https://math-comp.github.io/mcb/book.pdf.

[3] Jaap Boender, Florian Kammüller & Rajagopal Nagarajan (2015): *Formalization of Quantum Protocols using Coq*. In Chris Heunen, Peter Selinger & Jamie Vicary, editors: Proceedings of the 12th International Workshop on *Quantum Physics and Logic*, Oxford, U.K., July 15-17, 2015, *Electronic Proceedings in Theoretical Computer Science* 195, Open Publishing Association, pp. 71–83, doi:10.4204/EPTCS.195.6.

[4] Sylvie Boldo, Catherine Lelay & Guillaume Melquiond (2015): *Coquelicot*. http://coquelicot.saclay.inria.fr/.

[5] Guillaume Cano, Cyril Cohen, Maxime Dénès, Anders Mörtberg & Vincent Siles (2016): *CoqEAL - The Coq Effective Algebra Library*. https://github.com/CoqEAL/CoqEAL.

[6] Coq Development Team (2017): *The Coq Proof Assistant Reference Manual, Version 8.6*. Electronic resource, available from http://coq.inria.fr.

[7] Andrew Cross: *qasm-tools: An interoperable open-source software tool chain for studying fault-tolerant quantum circuits*.

[8] Ellie D'Hondt & Prakash Panangaden (2006): *Quantum weakest preconditions*. *Mathematical Structures in Computer Science* 16(03), pp. 429–451, doi:10.1017/S0960129506005251.

[9] Simon J. Gay, Rajagopal Nagarajan & Nikolaos Papanikolaou (2008): *QMC: A Model Checker for Quantum Systems*. In Aarti Gupta & Sharad Malik, editors: *Computer Aided Verification: 20th International Conference, CAV 2008*, Springer Berlin Heidelberg, Princeton, NJ, USA, pp. 543–547, doi:10.1007/978-3-540-70545-1_51.

[10] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *Quipper: A Scalable Quantum Programming Language*. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2013, pp. 333–342, doi:10.1145/2491956.2462177.

[11] IBM (2017): *IBM Quantum Experience*. Available at http://research.ibm.com/ibm-q/qx/.

[12] Xavier Leroy (2004): *The CompCert verified compiler*. *Development available at http://compcert. inria. fr* 2009.

[13] Michael A Nielsen & Isaac L Chuang (2010): *Quantum computation and quantum information*. Cambridge university press, doi:10.1017/CBO9780511976667.

[14] Jennifer Paykin, Robert Rand & Steve Zdancewic (2017): *QWIRE: A Core Language for Quantum Circuits*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, ACM, New York, NY, USA, pp. 846–858, doi:10.1145/3009837.3009894.

[15] Jennifer Paykin & Steve Zdancewic (2017): *The Linearity Monad*. In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, ACM, pp. 117–132, doi:10.1145/3122955.3122965.

[16] F. Pfenning & C. Elliott (1988): *Higher-order Abstract Syntax*. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, ACM, New York, NY, USA, pp. 199–208, doi:10.1145/53990.54010.

[17] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg & Brent Yorgey (2016): *Software Foundations*. Electronic textbook. Version 4.0. http://www.cis.upenn.edu/~bcpierce/sf.

[18] Mathys Rennela & Sam Staton (2017): *Classical control and quantum circuits in enriched category theory*. In: *Proceedings of the 33rd Conference on the Mathematical Foundations of Programming Semantics, MFPS*.

[19] Neil J. Ross (2015): *Algebraic and Logical Methods in Quantum Computation*. Ph.D. thesis, Dalhousie University.

[20] Peter Selinger & Benoît Valiron (2009): *Quantum lambda calculus*. In Simon Gay & Ian Mackie, editors: *Semantic Techniques in Quantum Computation*, Cambridge University Press, pp. 135–172, doi:10.1017/CBO9781139193313.005.

[21] Sam Staton (2015): *Algebraic Effects, Linearity, and Quantum Programming Languages*. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, ACM, New York, NY, USA, pp. 395–406, doi:10.1145/2676726.2676999.

[22] Dave Wecker & Krysta M Svore (2014): *LIQUiD: A software design architecture and domain-specific language for quantum computing*. *arXiv:1402.4467 [quant-ph]*.

[23] Mingsheng Ying (2011): *Floyd–hoare logic for quantum programs*. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33(6), p. 19, doi:10.1145/2049706.2049708.

[24] Mingsheng Ying, Shenggang Ying & Xiaodi Wu (2017): *Invariants of quantum programs: characterisations and generation*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ACM, pp. 818–832, doi:10.1145/3093333.3009840.