Abstract

This paper introduces QWIRE ("choir"), a language for defining quantum circuits and an interface for manipulating them inside of an arbitrary classical host language. QWIRE is minimal—it contains only a few primitives—and sound with respect to the physical properties entailed by quantum mechanics. At the same time, QWIRE is expressive and highly modular due to its relationship with the host language, mirroring the QRAM model of computation that places a quantum computer (controlled by circuits) alongside a classical computer (controlled by the host language).

We present QWIRE along with its type system and operational semantics, which we prove is safe and strongly normalizing whenever the host language is. We give circuits a denotational semantics in terms of density matrices. Throughout, we investigate examples that demonstrate the expressive power of QWIRE, including extensions to the host language that (1) expose a general analysis framework for circuits, and (2) provide dependent types.

Categories and Subject Descriptors D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs

Keywords quantum programming languages, quantum circuits, linear types, denotational semantics

1. Introduction

The standard architecture for quantum computers follows the quantum circuit model, which presents quantum computations as sequences of gates over qubits (the quantum analogue of bits). As with classical circuits, quantum circuits exist at a very low level of abstraction, and yet in spite of this, researchers and industry professionals write complex quantum algorithms in state-of-the-art quantum circuit languages like Quipper (Green et al. 2013a) and LIQUi|⟩ (Wecker and Svore 2014).

Why is the quantum circuit model so successful? In part, it is due to the fact that quantum data like qubits are extremely unintuitive from a classical perspective. Research into simple operations on quantum data, such as qubit-controlled conditionals and recursion, is still in its infancy (Ying 2014; Badescu and Panangaden 2015), so programmers cannot be sure that their algorithms using such abstractions are valid quantum-mechanically.

Although circuits manipulate quantum data, they themselves are classical data—a circuit is just a sequence of instructions describing how to apply gates to wires. In practice this means that circuits can be used to build up layers of abstractions hiding the low-level details. The QRAM model of quantum computing (Knill 1996) formalizes this intuition by describing how a quantum computer could work in tandem with a classical computer. In the QRAM model the classical computer handles the majority of ordinary tasks, while the quantum computer performs specialized quantum operations. To communicate, the classical computer sends instructions to the quantum machine in the form of quantum circuits. Over the course of execution, the quantum computer sends measurement results back to the classical computer as needed.

Although circuits manipulate quantum data, they themselves are classical data—a circuit is just a sequence of instructions describing how to apply gates to wires. In practice this means that circuits can be used to build up layers of abstractions hiding the low-level details. The QRAM model of quantum computing (Knill 1996) formalizes this intuition by describing how a quantum computer could work in tandem with a classical computer. In the QRAM model the classical computer handles the majority of ordinary tasks, while the quantum computer performs specialized quantum operations. To communicate, the classical computer sends instructions to the quantum machine in the form of quantum circuits. Over the course of execution, the quantum computer sends measurement results back to the classical computer as needed.

Embedded languages like Quipper, LIQUi|⟩, the Q language (Betteli et al. 2003), and the quantum IO monad (Altenkirch and Green 2010) can be thought of as instantiations of this model. They execute by running host language programs on the classical computer, making specialized calls to the (hypothetical) quantum machine. The classical host languages allow programmers to easily build up high-level abstractions out of low-level quantum operations. However, such abstractions are only worthwhile if the circuits they produce are safe—if they do not cause errors when executed on a quantum computer. Unfortunately, proving that an embedded language produces well-formed circuits is hard because it means reasoning about the entirety of the classical host language. This is frustrating when we care most about the correctness of quantum programs, which we expect to be both more expensive and error-prone than the embedded language’s classical programs.

One way of ensuring the safety of circuits is via a strong type system. Type safety for a quantum programming language means that well-formed circuits will not get stuck or “go wrong” when executed on a quantum machine. A subtlety is that this definition implies that the quantum program is even implementable on a quantum computer—that the high-level operational view of the language is compatible with quantum physics. One way of ensuring that the language is implementable is to give a denotational semantics for programs in terms of quantum mechanics.

Several quantum programming languages have been proposed with an emphasis on type safety, including Selinger’s QPL language (Selinger 2004), the quantum lambda calculus (Selinger and Valiron 2009), QML (Altenkirch and Grattage 2005), and ProtoQuipper (Ross 2015). However, these are toy languages, not designed for implementation in a conventional programming environment.
In this paper we address the tension between expressive embedded languages and denotationally-sound type-safe languages.

**The best of both worlds: QWIRE**

We propose the design of a core quantum circuit language in which circuits, equipped with a purely linear type system to ensure type safety, are explicitly separated from an arbitrary classical host language. The circuit language, which we call QWIRE ("choir"), comes equipped with an interface to this host language that allows for all the benefits of an embedded language while maintaining type safety and soundness.

The quantum lambda-calculus popularized the use of linear types for quantum systems. The “no-cloning” theorem of quantum mechanics states that quantum data cannot be cloned; in a programming environment, linear types ensure that quantum programs do not try to violate this property. However, the programming model should also allow for non-linear programming of ordinary classical data. The quantum lambda-calculus addresses this via subtyping, but for QWIRE we take an alternative approach inspired by the symmetry between the QRAM model and Benton’s Linear/Non-Linear (LNL) logic (1995):

![Diagram of Linear and Non-Linear Types]

In QWIRE, quantum circuits execute on the quantum computer and are given linear types, while host language programs execute on the classical computer and are given ordinary non-linear types.

Structuring the system in this way has several advantages. First, the interface to circuits is minimal, which means that they can be easily studied reasoned about. Second, the host language is extensible, since changes to the host language don’t induce changes to the circuit language, and vice versa. Third, the relationship between the circuit language and host language can be easily axiomatized: every circuit can be promoted to the host language via a box operator, and then unboxed to be reused inside of other circuits. This allows circuits to be treated as classical data structures in the host language, while prohibiting quantum data such as qubits from escaping the linear type system.

The axiomatic approach means that the circuit language is relatively independent from the host language. In particular, we expect that the host language could be instantiated with a wide range of programming languages depending on the intended use: high-level functional programming languages for developing and reasoning about algorithms; theorem provers for verification of quantum circuits; and perhaps even hardware description languages for deployment of real quantum computers.

**Contributions.**

- We present QWIRE, a core quantum circuit language, along with a simple linear type system (Section 3) and an equational operational semantics (Section 4). In addition to the circuit language itself, we describe a minimal interface to a classical host language that allows for modularity and communication via the QRAM model.
- We prove that the operational semantics of QWIRE is type safe (Theorems 6 and 7), and that all circuits reduce to a small set of normal forms (Theorem 8), depending only on the correctness of the host language.
- We give a denotational semantics in terms of density matrices (Section 5) and prove that the operational semantics is sound with respect to it (Theorem 11).
- Throughout we give examples of circuits written in an archetypal host language with access to QWIRE (Section 2). We also consider how to extend the host language with case analysis of circuits and dependent types (Section 6) to express programs that cannot be written in existing circuit languages.

**2. QWIRE by Example**

We start by taking a look at some code written in a host language that has access to QWIRE circuits. Circuits are constructed by a box operator that binds the input, represented as a wire name, inside of a circuit. Each wire name is identified with a wire type, which is either a bit, a qubit, or a (possibly empty) product of wire types.

For example, the identity circuit is written \( \text{id} = \text{box } w \Rightarrow \text{output } w \) and has the type \( \text{Circ} \left( W, W \right) \) for any wire type \( W \). The wire name \( w \) is not a regular variable as one would use in a classical programming language like the host language. For one, a wire is not first class: it is not by itself a circuit. For another, wire variables can only be used inside a circuit, and must be used linearly—once it is used, the same wire cannot be used again.

Gate application is the most important operation on wires. For example, the following circuit applies a Hadamard gate \( H \) to its input wire, followed by a measurement gate. Each gate has an associated input and output type and can only be applied to wires of the appropriate type.

```plaintext
hadamard-measure : \text{Circ}(qubit, bit) =
    box w =>
    u' <- gate H w;
    b <- gate meas u';
    output b
```

Note that we sometimes write \( \text{gate g w} \) as shorthand for the \((u' \leftarrow \text{gate g w}; \text{output u'})\) that appears in the example.

The reason wires must be treated linearly is that applying a gate changes the nature of the wire \( w \). It is meaningless to apply two gates to the same wire, because wires (and in particular qubits) cannot be duplicated. The following code, for example, is absurd:

```plaintext
absurd = box w =>
    x <- gate meas w;
    w' <- gate H w;
    output (x, w')
```

Similarly, it is dangerous to implicitly discard references to wires, which might be entangled in a greater quantum system. In QWIRE the discard gate explicitly discards a bit-valued wire, whereas qubit-valued wires must be measured before being discarded.

Since gates act on wires and not entire circuits, the expression \( \text{gate meas (gate H w)} \) is ill-formed. However, circuits can be composed by connecting the output of one circuit to the input of another. This type of composition is most useful when using circuits that have previously been constructed by a box operator. Boxed circuits can be unboxed by connecting some free input wires to the input of the box. The following function composes two boxed circuits in sequence, resulting in one complete circuit:

```plaintext
inSeq (c1 : Circ(W1, W2)) (c2 : Circ(W2, W3)) =
    box w1 =>
    w2 <- unbox c1 w1;
    unbox c2 w2;
```

The type system guarantees that the output wire of the first circuit matches the input wire to the second. More complex composi-
tion is also possible. For instance, \texttt{inPar} composes any two circuits in parallel, with no restriction on their wire types.

\begin{verbatim}
\texttt{inPar (c : Circ(W1,W2)) (c' : Circ(W1',W2')) : Circ(W1⊗W1', W2⊗W2') =
\begin{array}{l}
\texttt{box (w1,w1')} =>
\begin{array}{l}
u2 <- unbox c w1;
w2' <- unbox c' w1';
\end{array}
\begin{array}{l}
W_1 \quad W_2
\end{array}
\end{array}
\end{verbatim}

In the host language, we can write functions that compute circuits based on classical values, such as the following initialization function for qubits that determines which initialization gate gets applied.

\texttt{init (b : Bool) : Circ(1,qubit) =}
\begin{verbatim}
if b then box () => gate init1 ()
else box () => gate init0 ()
\end{verbatim}

\textbf{Quantum teleportation.} The quantum teleportation algorithm (adapted from Green et al.’s introduction to the Quipper language, 2013b) highlights the relationship between boxed and unboxed circuits more clearly. Figure 1 shows the quantum teleportation circuit, broken up into four parts. Alice is trying to send a qubit \texttt{q}, the input to the teleport circuit, to Bob. The circuit \texttt{bell00} initializes two qubits in the zero state (written \(|0\rangle\)), places qubit \texttt{a} in a superposition of \(|0\rangle\) and \(|1\rangle\) via the Hadamard \(|H\rangle\) gate, and entangles it with qubit \texttt{b} by applying a controlled-not (\texttt{CNOT}) gate. Qubit \texttt{a} is then given to Alice, and qubit \texttt{b} to Bob. Alice entangles \texttt{a} and \texttt{q} and measures them, outputting a pair of bits \texttt{x} and \texttt{y}. Bob then uses these to transform his own qubit into the state of the original qubit \texttt{q}.

\begin{verbatim}
\texttt{bob-dyn : Circ(bit⊗bit⊗qubit, qubit) =}
\begin{array}{l}
\texttt{box (w1,w2,q) =>}
\begin{array}{l}
(x1,x2) <- lift (w1,w2);
q <- unbox (if x2 then X_gate else id) q;
unbox (if x1 then Z_gate else id) q
\end{array}
\begin{array}{l}
\texttt{W} \quad \texttt{W} \quad \texttt{W} \quad \texttt{W}
\end{array}
\end{array}
\end{verbatim}

where \texttt{X_gate = box w => gate X w} and similarly for \texttt{Z_gate}.

On the one hand, dynamic lifting produces legible code that is easy to understand because it concentrates more computation in the host language. On the other hand, dynamic lifting is inefficient because the host language code must be run on a classical computer, during which time the quantum computer must remain suspended, waiting for the remainder of the circuit to be computed. Although dynamic lifting is not necessary in the case of quantum teleportation, it is an integral part of many quantum algorithms including quantum error correction, and so must be accounted for coherently.

The examples shown so far describe all of the ways to construct circuits in \texttt{QWIRE}. However, when describing quantum algorithms, circuits are ultimately intended to be executed on a quantum computer. The final piece of the story is therefore the run operation, which takes a circuit with no input and produces a value. For example, the following code implements a quantum coin toss:

\begin{verbatim}
\texttt{flip : Bool =}
\begin{array}{l}
\texttt{run (q <- gate init0 (); q <- gate H q; b <- gate meas q; output b)
\end{array}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{teleportation.png}
\caption{A \texttt{QWIRE} implementation of quantum teleportation.}
\end{figure}

\section{The \texttt{QWIRE} Circuit Language}

This section introduces the syntax and type theory of \texttt{QWIRE} and the interface for integrating \texttt{QWIRE} circuits into a host language.

\subsection{Circuit language}

As shown above, a circuit can be thought of as a sequence of gates on wires. These wires are described by their \texttt{wire type} \(W\), which is either unit (has no data), a bit or qubit, or a tuple of wire types.\footnote{Strictly speaking, the collection of wire types, along with the patterns for each wire type, could be thought of as an input to the system, provided that typing judgments for patterns are all syntax-directed. For example, we could consider a system without bit-valued wires, where measurement is only done via dynamic lifting. Alternatively we could consider more complex quantum data types in the style of Quipper (Green et al. 2013a). All wire types should be finite, however; see the discussion in Section 7.4 for more.}

\begin{verbatim}
W ::= 1 | bit | qubit | W_1 ⊗ W_2
\end{verbatim}

\begin{verbatim}
bell00 : Circ(1,qubit⊗qubit) =
\begin{array}{l}
\texttt{box () =>}
\begin{array}{l}
a <- gate init0 ();
b <- gate init0 ();
a <- gate H a;
gate CNOT (a,b)
\end{array}
\begin{array}{l}
\texttt{alice : Circ(qubit⊗qubit, bit⊗bit) =}
\begin{array}{l}
\texttt{box (q,a) =>}
\begin{array}{l}
(q,a) <- gate CNOT on (q,a)
x <- gate H q;
y <- gate meas q;
output (x,y)
bob : Circ(bit⊗bit⊗qubit, qubit) =
\begin{array}{l}
\texttt{box (x,y,b) =>}
\begin{array}{l}
(y,b) <- gate (bit-control X) (y,b);
(x,b) <- gate (bit-control Z) (x,b);
() <- gate discard y;
() <- gate discard x;
output b
teleport : Circ(qubit,qubit) =
\begin{array}{l}
\texttt{box q =>}
\begin{array}{l}
(a,b) <- unbox bell00 ();
(x,y) <- unbox alice (q,a);
unbox bob (x,y,b)
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{verbatim}
QWire is parameterized by a collection of gates $\mathcal{G}$, which each come equipped with input and output types. We write $\mathcal{G}(W_1, W_2)$ for the set of gates with input $W_1$ and output $W_2$. The gate set could consist of any collection of gates, but in the setting of quantum circuits it is conventional to choose a universal subset $\mathcal{U} \subseteq \mathcal{G}$ of unitary gates such that, for every $u \in \mathcal{U}(W, W)$, we also have

$$u^\dagger \in \mathcal{U}(W, W)$$
control $u \in \mathcal{U}(\text{qubit} \otimes W, \text{qubit} \otimes W)$
bit-control $u \in \mathcal{U}(\text{bit} \otimes W, \text{bit} \otimes W)$

Additionally, for the sake of this paper we assume we have initialization gates for bits and qubits:

new0, new1 $\in \mathcal{G}(1, \text{bit}), \text{init0, init1} \in \mathcal{G}(1, \text{qubit})$

as well as a measurement gate meas $\in \mathcal{G}(\text{qubit}, \text{bit})$ and a discard gate discard $\in \mathcal{G}(\text{bit}, 1)$ for bits.

A typing judgment $\Gamma; \Omega \vdash C : W$ specifies when a circuit is well-formed. In this judgment,

• $C$ is a circuit;
• $\Omega = w_1 : W_1, \ldots, w_n : W_n$ is a context of input wire names with their wire types;
• $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is a context of host language variables with their host language types; and
• $W$ is the output type of the circuit.

Thus, all well-typed circuits have the following shape:

\[
\begin{array}{c}
\Omega \\
\end{array} 
\begin{array}{c}
C \\
\end{array} 
\begin{array}{c}
W \\
\end{array}
\]

Wires in QWire are linear, which means that they cannot be duplicated or discarded,\footnote{Of course, gates may exist that duplicate or discard bits, but wires themselves are linear structures.} and when we write $\Omega, \Omega'$ we assume that $\Omega$ and $\Omega'$ contain only disjoint wire names. Both $\Omega$ and $\Gamma$ are thought of as unordered contexts.

The output of a circuit is built up as a pattern of its input wires:

\[
\Omega \Rightarrow p : W \\
\Gamma; \Omega \vdash \text{output } p : W \\
\]

A pattern is just a tuple of wires identifying a single wire type.

\[
\Omega \Rightarrow (\cdot) : 1 \\
\Omega \Rightarrow w : W \\
\Omega \Rightarrow w : W \\
\Omega_1 \Rightarrow p_1 : W_1 \\
\Omega_2 \Rightarrow p_2 : W_2 \\
\Omega_1, \Omega_2 \Rightarrow (p_1, p_2) : W_1 \oplus W_2 \\
\]

A gate can be applied to a pattern of wires when permitted by the signature of the gate. The output of that gate is then decomposed by another pattern. The wires exiting the gate can then be used in the remainder of the circuit.

\[
g \in \mathcal{G}(W_1, W_2) \\
\Omega_1 \Rightarrow p_1 : W_1 \\
\Omega_2 \Rightarrow p_2 : W_2 \\
\Gamma; \Omega_1, \Omega_2 \vdash (g, p_1, p_2) : W_1 \oplus W_2 \\
\Gamma; \Omega, \Omega_1, \Omega_2 \vdash g : C : W \\
\]

We compose circuits by connecting the output of one circuit to the input wires of another. This operation differs from sequential composition in that the second circuit may have additional inputs.

\[
\Gamma; \Omega_1 \vdash C : W \\
\Gamma; \Omega_2 \vdash p : W \\
\Gamma; \Omega_1, \Omega_2 \vdash C' : W' \\
\Gamma; \Omega \vdash p \leftarrow C; C' : W' \\
\]

\[
\begin{array}{c}
\Omega_1 \\
C \\
\Omega \\
\end{array} 
\begin{array}{c}
\Omega \\
\Omega \\
\end{array} 
\begin{array}{c}
W \\
C' \\
W' \\
\end{array}
\]

**3.2 Host language**

In the QRAM model, a classical computer works together with a quantum computer. The classical computer communicates with the quantum computer by sending it instructions—that is, circuits in QWire. Terms in the host language, meanwhile, describe computations on the classical computer. We refer to the host language as HOST and describe some of its properties.

We assume that HOST is statically typed, and write its types as $A$. Furthermore, we assume that for each wire type there is a corresponding classical type—for example, a host-level boolean might correspond to the qubit and bit wire types, and tensor wire types correspond to pairs. In addition, we add a type representing the QWire circuits between two wire types, which we write Circ($W_1, W_2$). Of course, HOST will often contain many other types, including functions and inductive data types, but the interface with QWire does not depend on the particular structure of HOST. For this reason we say that HOST is arbitrary: many different languages could fill in for the host language of QWire.

Overall, we can summarize the types of HOST as follows:

\[
A ::= \cdots | \text{Unit} | \text{Bool} | A \times A | \text{Circ}(W_1, W_2)
\]

The typing judgment for HOST terms is written $\Gamma \vdash t : A$ where $\Gamma$ is a context of variables with their associated types.

**Boxing and Unboxing.** The Circ type bridges QWire circuits and HOST types. The type Circ($W_1, W_2$) is a wrapper around QWire circuits of the form $\Gamma; \Omega \vdash C : W_2$, where the wires in $\Omega$ come from a pattern destructing the input type $W_1$.

\[
\begin{array}{c}
\Omega \Rightarrow p : W_1 \\
\Gamma; \Omega \vdash C : W_2 \\
\Gamma \vdash \text{box } (p : W_1) : \text{Circ}(W_1, W_2) \\
\end{array} 
\begin{array}{c}
\text{Circ} \\
W \\
\end{array} 
\begin{array}{c}
\Omega \\
\end{array} 
\begin{array}{c}
W_2 \\
\end{array}
\]

A boxed term of type Circ($W_1, W_2$) can be coerced back into a QWire circuit by describing how to match up the available input wires to the input type of the boxed representation.

\[
\Gamma \vdash t : \text{Circ}(W_1, W_2) \\
\Gamma; \Omega \vdash p : W_1 \\
\Gamma \vdash \text{unbox } t : p : W_2 \\
\]

**Lifting.** In the QRAM model described above, the quantum computer also communicates with the classical computer by sending it the results of measurement. For example, given a circuit with no input wires and a bit output, running that circuit should result in a host language boolean value.

\[
\Gamma; \vdash C : \text{bit} \\
\Gamma; \vdash \text{run } C : \text{Bool}
\]

We can generalize this operation so that running a circuit that outputs a qubit implicitly measures that qubit and returns the corresponding boolean. In fact this relationship generalizes to any wire type, which can be lifted to a classical type as follows:

\[
|\text{bit} | = \text{Bool} \\
|\text{1} | = \text{Unit} \\
|\text{qubit} | = \text{Bool} \\
|W_1 \otimes W_2 | = |W_1| \times |W_2|
\]
The run operator now has the following form:

\[
\Gamma; \vdash C : W \\
\Gamma \vdash \text{run } C : [W]
\]

Run is a static lifting operator, meaning that there is no residual state left on the quantum computer after run \( C \) has completed. In contrast, dynamic lifting describes the case when, over the course of a quantum computation, a subset of the wires are measured and communicated to the classical computer. In this case, the classical computer must continuously undergo error correction to prevent degradation. However, dynamic lifting is a fundamental form of communication between the two machines, and is needed to implement algorithms like quantum error correction.

We write the dynamic lifting operator \( x \leftarrow \text{lift } p ; C \) to mean that the wires in \( p \) are measured, lifted to the classical computer as the host variable \( x \), and used to compute the circuit \( C \).

\[
\Omega \Rightarrow p : W \\
\Gamma, x : [W]; t' \vdash C : W'
\]

\[
\Gamma; \Omega, \Omega' \vdash x \leftarrow \text{lift } p ; C : W'
\]

The dynamic and static lifting operations are not mutually derivable, as they represent two fundamentally different ways to communicate the results of measurement between the two systems.

### 3.3 Static semantics

To summarize, the syntax of \( \text{QWIRE} \) circuits and host terms include the following:

- **(Patterns)** \( p ::= () | w | (p, p) \)
- **(Circuits)** \( C ::= \text{output } p | p_2 \leftarrow \text{gate } g \ p_1; C | p \leftarrow C; C \)
- **(Terms)** \( t ::= \cdots \cdot \cdot \cdot \vdash \text{run } C | \text{box } (p : W) \Rightarrow C \)

The typing rules are summarized in Figure 2. Note that we often write box \( p \Rightarrow C \) instead of box \( (p : W) \Rightarrow C \) when the type of the input pattern is clear. Note that typing contexts are unique for both patterns and circuits.

**Lemma 1.** If \( \Omega_1 \Rightarrow p : W \) and \( \Omega_2 \Rightarrow p : W \) then \( \Omega_1 = \Omega_2 \). If \( \Gamma; \Omega_1 \vdash C : W \) and \( \Gamma; \Omega_2 \vdash C : W \) then \( \Omega_1 = \Omega_2 \).

### 4. Operational semantics: circuit normalization

Circuits in \( \text{QWIRE} \) represent instructions to be executed on a quantum computer: either apply a particular gate, or request a dynamic lifting operation. Composition and unbox operations are more like meta-operations: they describe ways to construct more complex combinations of gates. In this section we define an operational semantics that eliminates all instances of unboxing and composition, resulting in a small set of normal forms. The subset of \( \text{QWIRE} \) circuits in normal forms are identified by two main properties.

First, normal circuits should operate only on bits and qubits, not on the tuples of wires described by arbitrary wire types \( W \). We call a circuit **concrete** when all of its input wires are either bits or qubits:

\( ; Q \vdash C : W \) where \( Q ::= \cdot | Q, w : \text{bit} | Q, w : \text{qubit} \).

A concrete circuit is called **normal** when it consists only of gate applications, outputs, and dynamic lifting operations.

\( N ::= \text{output } p | p_2 \leftarrow \text{gate } g \ p_1; N | x \leftarrow \text{lift } p ; C \)

Notice that the lifting operator \( x \leftarrow \text{lift } p ; C \) does not assume that its continuation \( C \) is also normal. This is because \( C \) has a free host-level variable \( x \) that cannot in general be normalized. For example, consider the circuit \( x \leftarrow \text{lift } w ; \text{unbox } (\text{init } x) () \); the continuation \( \text{unbox } (\text{init } x) () \) cannot be normalized because \( \text{init } x \) does not reduce in the host language.

In the rest of this section we define the small-step operational semantics that reduces concrete circuits typed by \( ; Q \vdash C : W \) to normal circuits. The operational rules rely on a fairly complex substitution relation, which we briefly address.

#### Substitution

A substitution \( \{ p' / p \} \) describes a finite map from wire names to patterns. It is well-defined only when \( p \) generalizes \( p' \) (written \( p \gg p' \)) in the following sense:

\[
p' \gg w \iff (p') \gg (w) \iff \{ p' \gg w \} = \{ p \gg w \}
\]

\[
p' \gg w \iff \{ p' \gg w \} = \{ p \gg w \}
\]

We say \( p' \gg p \) when \( p' \gg p \) and \( \neg (p \gg p') \), and we say \( p \) is concrete for \( W \) when, for all \( \Omega \Rightarrow p' : W \), \( \neg (p' \gg p) \).

**Lemma 2.** If \( \Omega \Rightarrow p : W \) and \( Q \Rightarrow p' : W \), then \( p' \gg p \).

The substitution map is defined as follows:

\[
\{ () / () \} = \emptyset \\
\{ p' / w \} = w \mapsto p' \\
\{ (p_1, p_2) / (p_1, p_2) \} = \{ p_1' / p_1 \}, \{ p_2' / p_2 \}
\]

A well-defined substitution extends to **total** functions on patterns, circuits, and wire contexts. For patterns, we have:

\[
\{ () / (p' / p) \} = () \\
w \{ p' / p \} = \begin{cases} p_0 & \text{if } w \mapsto p_0 \in \{ p' / p \} \\ w & \text{otherwise} \end{cases}
\]

\[
\{ p_1, p_2 \} \{ p' / p \} = \{ p_1 \{ p' / p \}, p_2 \{ p' / p \} \}
\]
The operation on circuits is straightforward, assuming the usual notions of capture-avoidance.

\[
\begin{align*}
\text{(output } p_0) & \{p/p’\} = \text{output } (p_0(p’/p)) \\
(\text{gate } p_1; C) & \{p/p’\} = p_2 \rightarrow \text{gate } g p_1(p’/p); \ C \{p’/p\} \\
(x \leftarrow \text{lift } p_0; C) & \{p/p’\} = x \leftarrow \text{lift } p_0(p’/p); \ C \{p’/p\} \\
(\text{unbox } t p_0) & \{p/p’\} = \text{unbox } t (p_0(p’/p)) \\
(p_0 \rightarrow C; C’) & \{p/p’\} = p_0 \rightarrow C \{p’/p\}; \ C’ \{p’/p\}
\end{align*}
\]

A well-defined substitution \{p’/p\} is consistent with \omega at \ W if \ (w, p_0) \in \{p’/p\} implies that there is some (unique) \Omega_0 such that \Omega_0 \Rightarrow p_0 : \ W. A substitution is consistent with a context \Omega when, for all \omega : \ W \in \Omega, it is consistent with \omega at \ W.

For wire contexts, suppose \{p’/p\} is consistent with \Omega. The substitution \Omega \{p’/p\} is defined by induction on \Omega:

\[
\{p/p’\} = \begin{cases}
\emptyset & \text{if } w \mapsto p_0 \in \{p’/p\} \\
\Omega’ \{p’/p\}, \ W : w’ & \text{otherwise}
\end{cases}
\]

![Box](Box)
Lemma 5. If p is concrete for W then there is a unique Q such that Q ⇒ p : W. Furthermore, for every wire type W there exists a p (not necessarily unique) such that p is concrete for W.

Since an unbox operator is not a normal circuit, we eliminate it via a β rule once its argument t reaches a value of the form box p ⇒ N. Similarly, the composition operator reduces its first argument to a normal form before taking a step. When the argument is an output-output p′, the composition p ←→ output p′; C uses substitution to take a β-reduction step. On the other hand, when the argument consists of gate or lifting step, the semantics commutes that command to the front of the circuit; we call these operators commuting conversions.

4.1 Type safety.
We prove type safety with progress and preservation theorems, provided that the relation −→H is also type safe.

Theorem 6 (Preservation). Suppose −→H satisfies preservation.
1. If ⊢ t : A and t −→ t′, then ⊢ t′ : A.
2. If ; Q ⊢ C : W and C ⇒ C′, then ; Q ⊢ C′ : W.

Proof. By induction on the step relation (Appendix A).

Theorem 7 (Progress). Suppose −→H satisfies progress with respect to the values vθ.
1. If ⊢ t : A then either t is a value vθ or there is some t′ such that t −→ t′.
2. If ; Q ⊢ C : W then either C is normal or there is some C′ such that C ⇒ C′.

Proof. By induction on the typing judgment (Appendix A).

Provided that −→H is strongly normalizing, we can also show that circuits are strongly normalizing.

Theorem 8 (Normalization). Suppose that −→H is strongly normalizing with respect to vθ.
1. If ⊢ t : A, there exists some value vθ such that t −→∗ vθ.
2. If ; Q ⊢ C : W, there exists some normal circuit N such that C ⇒∗ N.

Proof. By induction on the number of constructors in the term and circuit (Appendix A).

5. Denotational Semantics
In this section we will give a denotational semantics for QWIRE circuits. The state of a quantum system can be described in terms of a density matrix, in which numbers along the diagonal represent the probability of measuring a given state. Consider, for instance, the entangled Bell pair produced by the following circuit:

\[
\begin{pmatrix}
|0\rangle & |1\rangle \\
|1\rangle & |0\rangle
\end{pmatrix}
\]

This pair of qubits is represented by the density matrix
\[
\begin{pmatrix}
\frac{1}{2} & 0 & 0 & \frac{1}{2} \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & 0 & \frac{1}{2}
\end{pmatrix}
\]

where the \(\frac{1}{2}\) in the top left represents the probability of measuring two zeros, while the \(\frac{1}{2}\) in the bottom right represents the probability of measuring two ones. If we measured this system, we would obtain the mixed state density matrix
\[
\begin{pmatrix}
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2}
\end{pmatrix}
\]

representing a distribution over \(|00\rangle\) and \(|11\rangle\).

Since a QWIRE circuit transforms some state to another, it will be interpreted as a superoperator over density matrices. In the rest of this section we will assume some familiarity with the mathematics of quantum computation; for reference we encourage readers to consult Nielsen and Chuang’s standard text in the area (2010).

Given a Hilbert space \(H\), we write \(H^*\) for the collection of density matrices seen as linear transformations from \(H\) to \(H\). Given a linear map on Hilbert spaces \(f : H \rightarrow H^*\), \(f^*\) is a superoperator from \(H^*\) to \((H^*)^*\) defined by \(f^* \rho = f \rho^f\). In fact, every superoperator can be written
\[
\Phi \rho = \sum_{i \in X} M_i \rho
\]

for some indexed family of matrices \(\{M_i\}_{i \in X}\). We define
\[
(\Phi \otimes \Phi') \rho = \sum_{(i,j) \in X \times X'} (M_i \otimes M_j') \rho.
\]

In this model, a wire type is interpreted as a Hilbert space in the following way:

- \([\text{bit}] = H_2 = \{ |0\rangle, |1\rangle \}
- \([\text{qubit}] = H_2 = [W_1] \oplus [W_2] = [W_1] \\ [W_2]

The intention is that a circuit from \(W_1\) to \(W_2\) is interpreted as a superoperator mapping density matrices corresponding to \(W_1\) to density matrices corresponding to \(W_2\). Notice that the denotation of bit and qubit are identical, which reflects the fact that bit-valued wires on a quantum machine are implemented using qubits in the \(|0\rangle\) and \(|1\rangle\) states.

For example, every gate \(g \in G(W_1, W_2)\) be interpreted as a superoperator between \(W_1\) and \(W_2\). Although the set of gates is a parameter of the system, a unitary gate \(U\) should clearly correspond to \(U^*\), and the interpretation of other likely gates is as follows:

- \([\text{new0}]\), \([\text{init0}]\) = \(|0\rangle \langle 0|\)*
- \([\text{new1}]\), \([\text{init1}]\) = \(|1\rangle \langle 1|\)*
- \([\text{meas}]\) = \(|0\rangle \langle 0|\)* \(+ |1\rangle \langle 1|\)*
- \([\text{discard}]\) = \(|0\rangle^* \langle 1|\)*

QWIRE circuits are specified by an unordered context of input wires \(\Omega\). However, we can equally well think of \(\Omega\) as an ordered context, along with an explicit permutation rule to change the order of the wires.\(^6\)

\[
\begin{array}{c}
\Gamma; \Omega' \vdash C : W \\
\pi : \Omega \equiv \Omega' \\
\end{array}
\]

\(\Gamma; \Omega \vdash C : W\)

\(^3\)Formally, a density matrix is a positive Hermitian matrix whose trace sums to 1. Any pure state in column vector form can be transformed into a density matrix by taking its outer product with itself.

\(^6\)We elide these details in Section 3 as they complicate the operational semantics.
Permutations are defined inductively.

\[ \epsilon : \Omega \equiv \Omega \]

\[ \pi_1 : \Omega_1 \equiv \Omega_2 \quad \pi_2 : \Omega_2 \equiv \Omega_3 \]

\[ (\text{swap} \, \Omega_1 \, \Omega_2) : \Omega, \Omega_1, \Omega_2, \Omega' \equiv \Omega, \Omega_2, \Omega_1, \Omega' \]

Note that permutations are reflected in the typing judgments of circuits but not in the syntax. We extend the substitution relation to permutations in a natural way, writing \( \pi \{ p'/p \} \).

\[ \epsilon \{ p'/p \} = \epsilon \]

\[ (\pi_2 \circ \pi_1) \{ p'/p \} = \pi_2 \{ p'/p \} \circ \pi_1 \{ p'/p \} \]

\[ \text{swap} \, \Omega_1, \Omega_2 \{ p'/p \} = \text{swap} \, (\Omega_1 \{ p'/p \}) \, (\Omega_2 \{ p'/p \}) \]

An ordered context of wires is now interpreted as a Hilbert space by treating the comma as the tensor product:

\[ [(\cdot)] = H_1 \quad [(w : W)] = [W] \quad [(\Omega_1, \Omega_2)] = [\Omega_1] \otimes [\Omega_2] \]

Although the context of wires can be permuted inside a circuit, it will not be permuted inside a pattern. Therefore, a pattern \( \Omega \Rightarrow p : W \) is just a reassociation of the input wires; all permutations must be done outside the pattern. This means that whenever \( \Omega \Rightarrow p : W \), it must be the case that \( [\Omega] = [W] \).

A permutation \( \pi : \Omega \equiv \Omega' \) will be interpreted as a linear isomorphism from \( [\Omega] \) to \( [\Omega'] \), written \( [\pi] \), as follows:

\[ [\epsilon] = I \]

\[ [\pi_2 \circ \pi_1] = [\pi_2] \circ [\pi_1] \]

\[ [\text{swap} \, \Omega_1 \, \Omega_2] (v_1 \otimes v_2 \otimes v_3) = (v_1 \otimes v_2 \otimes v_1 \otimes v_3) \]

**Lemma 9.** If \( \pi : \Omega \equiv \Omega' \) and \( \{ p'/p \} \) is consistent with \( \Omega \), then \( [\pi \{ p'/p \}] = [\pi] \).

**Proof.** Straightforward by induction on the permutation.

For \( \cdot \vdash v : [W] \), we define \( [(v : [W])] \) to be an element of \( [W] \):

\[ [(\cdot : \text{Unit})] = [*] \]

\[ [(\text{false} : \text{Bool})] = [0] \]

\[ [(\text{true} : \text{Bool})] = [1] \]

\[ [(v_1, v_2) : [W_1] \times [W_2]] = [v_1 : [W_1]] \otimes [v_2 : [W_2]] \]

Now, for \( ; : \Omega \vdash C : W \), we write \( [\Omega \vdash C : W] \) for its interpretation as a superoperator between \( [\Omega]^* \) and \( [W] \). Furthermore, for \( ; : t : \text{Circ}(W_1, W_2) \), we write \( [t] \) for \( [\Omega \vdash C : W_2] \). As \( t \rightarrow_{\text{R}} \) box \( p \Rightarrow C \) in the host language and \( \Omega \Rightarrow p : W_1 \), this operation is functional exactly when the host language semantics is strongly normalizing.

The interpretation of circuits is defined in Figure 4.

**Theorem 11 (Soundness).** If \( ; : \Omega \vdash C : W \) and \( \Omega \Rightarrow C' \) then

\[ [\Omega \vdash C : W] = [\Omega \vdash C' : W] \]

**Proof.** By induction on the typing judgment (Appendix B).
The addition of the patterns now normalize just like circuits do. In particular, normal pattern typing judgment must include host-language variables, and a host-level representation of gates is straightforward: unpacked in a way similar to unboxing:

$\hat{\omega} \vdash \text{output } p : W$

$\vdash \omega \vdash C : W \quad \pi : \hat{\omega} \equiv \omega'$

$\vdash \omega \vdash C : W$

\[ g \in G(W_1, W_2) \]

$\hat{\omega} \Rightarrow p_1 : W_1 \quad \hat{\omega} \Rightarrow p_2 : W_2$\quad $\hat{\omega}_2, \hat{\omega} \vdash C : W$

$\vdash \omega \vdash p : W \quad x : \hat{\omega}; \omega' \vdash C : W'$

$\vdash \omega, \omega' \vdash x \leftarrow \text{lift } p; C : W'$

$\vdash \hat{\omega}, \hat{\omega}_2 \vdash p \leftarrow C; C' : W'$

\[ [\omega \vdash \text{output } p : W] = \hat{\omega}^* \]

$[\omega \vdash C : W] = [\omega' \vdash C : W] \circ [x]^*$

$[\omega \vdash \text{unbox } t \vdash p : W'] = [t : \text{Circ}(W, W')]$

$[\omega_1, \omega \vdash p_2 \leftarrow \text{gate } g \vdash p_1; C : W] = [\omega_2, \omega \vdash C : W] \circ ((g) \odot \hat{\omega}^*)$

$[\omega, \omega' \vdash x \leftarrow \text{lift } p; C : W'] = \sum_{x : \hat{\omega}} [\omega' \vdash C(\hat{\omega}/x) : W'] \circ ([x : \hat{\omega}] \odot \hat{\omega}^*)$

$[\omega_1, \omega_2 \vdash p \leftarrow C; C' : W'] = [\omega_0, \omega_2 \vdash C' : W'] \circ ([\omega_1 \vdash C : W] \odot \hat{\omega}^*)$

\[ \omega \Rightarrow \Omega \quad \Omega \Rightarrow p : W_1\]

\[ g \in \mathcal{G}(W_1, W_2) \]

$\hat{\omega} \Rightarrow \hat{\omega}_1 \leftarrow g \vdash \text{Gate}(W_1, W_2) \quad \hat{\omega}_2 \vdash p : W_2$

We can write the function that embeds an inductive circuit into a circuit:

$\text{fromICirc} \left( t : \text{ICirc}(W_1, W_2) \right) : \text{Circ}(W_1, W_2) = \left\{ \begin{array}{ll}
\text{case } t \text{ of} & \\
\text{Output } p \rightarrow \text{box } w_1 \Rightarrow \text{output } (\text{unpat } p \vdash w_1) & \\
\text{Gate } g \vdash c \rightarrow & \\
\text{box } (\text{unpat } (\text{reverse } p) (w_1, w_0)) \Rightarrow w_2 \leftarrow \text{gate } g \vdash w_1; & \\
\text{unbox } c (w_2, w_0) & \\
\text{Lift } f \rightarrow & \\
\text{box } (\text{unpat } (\text{reverse } p) (w, w')) \Rightarrow x \leftarrow \text{lift } v; & \\
\text{unbox } (f x) w' & 
\end{array} \right.$

\section*{Patterns.}

A host-level pattern can be constructed in a similar way to a host level circuit: we write $\hat{\omega} \Rightarrow p \vdash p'$ and think of it as a function between wire types. Host-level patterns can also be unpacked in a way similar to unboxing:

$\Gamma \vdash \hat{\omega} \Rightarrow \hat{\omega}_1 \vdash \hat{\omega}_1 \leftarrow g \vdash \hat{\omega}_2 \vdash \text{Gate}(W_1, W_2)$

We can show that $\omega$ and $\hat{\omega}$ are concrete. Then:

$\text{reverse } p \Rightarrow \omega \vdash \text{unpat } p : W_1$

$\Rightarrow \omega \vdash \text{unpat } p : W_1$

$\Rightarrow \omega \vdash \text{unpat } p : W_1$.

The addition of the $\text{Pat}(W_1, W_2)$ type means a few things: the pattern typing judgment must include host-language variables, and patterns now normalize just like circuits do. In particular, normal patterns are any of the form

\[ n ::= () \mid w \mid (n_1, n_2) \]

The progress and preservation theorems for patterns fall out naturally from the substitution lemma (Lemma 4).

We can reverse a host-level pattern in the following way:

$\text{reverse } p : \text{Pat}(W_1, W_2) = \text{pat}(\text{unpat } p : W_1)$

$\Rightarrow \omega \vdash \text{output } p : W$

We can show that $\text{reverse } p$ is $p$.

\section*{Gates.}

A host-level representation of gates is straightforward:

$\text{gate } g \vdash \text{Gate}(W_1, W_2)$

We expect a small number of operations on host-level gates, e.g.

$\text{isUnitary} : \text{Gate}(W_1, W_2) \rightarrow \text{Bool}$

$\text{transpose} : \text{Gate}(W_1, W_2) \rightarrow \text{Option } \text{Gate}(W_2, W_1)$

The addition of the $\text{Pat}(W_1, W_2)$ type means a few things: the pattern typing judgment must include host-language variables, and patterns now normalize just like circuits do. In particular, normal patterns are any of the form

$\text{unpat} (\text{pat } p_1 \Rightarrow p_2) \Rightarrow p_2 (p/p_1)$

Again, in order for substitution to be valid, it must be the case that the underlying pattern is concrete, for example:

$\text{pat}(p_1 : W) \Rightarrow p_2 \iff \text{pat} p_1' \Rightarrow p_2 (p_1/p_1)$

when $p_1'$ is concrete for $W$ and $p_1' \prec p_1$.

The progress and preservation theorems for patterns fall out naturally from the substitution lemma (Lemma 4).

We can reverse a host-level pattern in the following way:

$\text{reverse } p : \text{Pat}(W_1, W_2) = \text{pat}(\text{unpat } p : W_1)$

We can show that $\text{reverse } p$ is $p$. Suppose $p = (\text{pat } p_1 \Rightarrow p_2)$ where both $p_1$ and $p_2$ are concrete. Then:
The function from $\text{Circ}(W_1, W_2)$ to $\text{ICirc}$ $W_1, W_2$ is loosely the algorithm described above, and has the type signature

\[
\text{toICirc} \ (t : \text{Circ}(W_1, W_2)) = \text{ICirc} \ W_1, W_2
\]

However, toICirc is not expressible directly in the host language, since it relies on induction on the typing structure of circuits. Instead we describe it in the meta-theory. If $Q \Rightarrow p : W_1$ and $; Q \Rightarrow N : W_2$ then toICirc (box $p \Rightarrow N$) is defined on the typing structure of $N$, as shown in Figure 5.

**Theorem 12.** For all terms $t$ of type $\text{ICirc} W_1, W_2$ and $c$ of type $\text{Circ}(W_1, W_2)$, we have:

\[
\text{toICirc} \ (\text{fromICirc} \ t) = t
\]

\[
\text{fromICirc} \ (\text{toICirc} \ c) = c
\]

**Proof.** By induction on the typing judgment (Appendix C). □

**Reversing circuits.** The circuit reversal function can be written by interfacing with the ICirc type.

\[
\text{reverse} \ (c : \text{Circ}(W_1, W_2)) = \text{case \ toICirc} \ c \ of
\]

\[
\text{case} \ \text{fromICirc} \ (\text{Output} \ (\text{reverse} \ \text{pat} \ p)) \text{ \ with} \ (\text{Output} \ p) \Rightarrow \text{Gate} \ g \Rightarrow \text{reverse} \ \text{pat} \ p \ q \ \text{in} \ \text{inSeq} \ c \ \text{rev} \ \text{toICirc} \ i \ \text{rev}
\]

\[
\text{case} \ \text{reverse} \ (\text{toICirc} \ c'), \ \text{reverse} \ \text{gate} \ g \ \text{of}
\]

\[
\text{case} \ \text{reverse} \ \text{pat} \ c' \Rightarrow \text{Some} \ c \Rightarrow \text{Some} \ g \Rightarrow \text{let} \ p \ \text{rev} = \text{reverse} \ \text{pat} \ p \ \text{in}
\]

\[
\text{let} \ i \ \text{rev} = \text{Gate} \ \text{id} \ \text{pat} \ g \ \text{rev} \ (\text{Output} \ p \ \text{rev}) \ \text{in}
\]

\[
\text{inSeq} \ c \ \text{rev} \ (\text{fromICirc} \ i \ \text{rev})
\]

\[
\text{case} \ \text{reverse} \ \text{pat} \ c' \Rightarrow \text{None} \Rightarrow \text{None}
\]

\[
\text{case} \ \text{reverse} \ \text{gate} \ g \Rightarrow \text{None} \Rightarrow \text{None}
\]

where $\text{id} \ \text{pat} = \text{pat} u \Rightarrow u$ and inSeq is the sequential composition operator defined in Section 2. We also assume the existence of an operation reverse_gate on gates that is semantically valid, so that if reverse_gate $g$ is Some $g'$, then

\[
[g] \circ [g'] = [g'] \circ [g].
\]

In that case, we can prove the following correctness condition:

**Theorem 13.** If $\text{reverse} \ c = \text{Some} \ c'$ then

\[
[c] \circ [c'] = [c'] \circ [c] = [c'] = [c']
\]

**Proof.** By induction on terms (Appendix D). □

Other operations expressible in the host language with case analysis include:

- Less naive circuit reversal algorithms; for example qubit initialization can be reversed and treated as an ancilla if every operation following initialization can be reversed;
- Special purpose quantum simulators;
- A safe control operator on circuits that adds a control wire to every unitary gate and outputs None if it encounters a lift or non-unitary gate;
- Resource analyzers that count the number of gates in a circuit up to a dynamic lifting operation;
- An optimizer that collects the gates in a circuit into a data structure, runs an optimization pass, and reconstructs the circuit;
- A transformation that maps one set of unitary gates to another;
- A static analysis tool to determine whether two circuits are equivalent (Staton 2015).

Another way to gain expressivity of circuits is by adding dependent types to the host language.

---

**6.2 Dependent types**

Consider the quantum Fourier transform, which is a circuit with $n$ inputs and $n$ outputs. It is natural for the wire types of the Fourier circuit reflect this dependency on $n$. In the language of dependent types, it might have the signature

\[
\text{fourier} : \Pi \ (n : \text{Nat}). \ \text{Circ} (\otimes n \ \text{qubit}, \otimes n \ \text{qubit})
\]

where $\otimes$ is a type-level function that duplicates the argument type (qubit) some number of times (defined below).

Combining linear and dependent types is still an area of active research (Krishnaswami et al. 2015; McBride 2016) but thanks to the separation between the circuit and host languages, we can get away with a limited form of dependent types due to Krishnaswami et al. (2015). Under this strategy, types can depend on terms, but only terms of classical (non-linear) type. These include dependencies on wire types themselves, which are considered classical terms in the universe hierarchy.

To be more precise, let $W$ be the kind of wire types, and consider an indexed hierarchy of host language types $A_i$. We define a kind $\Pi_t$ with the following well-formedness judgment:

\[
\Gamma \vdash \Pi_t (x : A) : W
\]

In addition, we introduce a new host-language type $\Pi (x : A). B$ with the following well-formedness condition:

\[
\Pi (x : A). B : A_i \vdash \Pi (x : A). B : A_i
\]

$\Pi$ types have the usual introduction and elimination rules:

\[
\Gamma \vdash t : \Pi (x : A). A \quad \Gamma \vdash t' : A_i
\]

\[
\Gamma \vdash \Pi (x : A). A \quad \Pi (x : A). A \equiv \Pi (x : A'). A_i
\]

$\Pi$ types are not expressible directly in the host language, but are considered classical terms in the universe hierarchy.

**A dependent QFT.** Under this framework, we can start with the type-level function $\text{tensor}$:

\[
\text{tensor} (n : \text{Nat}) (W : \mathcal{L}) : \mathcal{L} =
\]

\[
\text{case} \ n \ \text{of}
\]

\[
| 0 \Rightarrow 1
\]

\[
| 1 \Rightarrow W
\]

\[
\text{let} \ n' => W \otimes \text{tensor} n' W
\]

We write $\otimes n W$ for tensor $n W$.

Next we use these length-indexed tuples to write a dependently-typed quantum Fourier transform in the style of Green et al. (2013b). Our version of the Fourier circuit ensures that the number of qubits in the input and output are always the same.

First, we define the rotation circuits. We assume the presence of a family of gates $\text{RGate}$ $n$ that rotates its input along the $z$-axis by $2\pi/n$ (Green et al. 2013b). The rotations circuit takes two natural number inputs: $m$, the argument given to the controlled $R$ gates; and $n$, the number of bits in the input.

\[
\text{rotations} (m : \text{Nat}) = \Pi (n : \text{Nat}).
\]

\[
\text{CIRC} (\otimes (n+1) \text{qubit}, \otimes (n+1) \text{qubit}) =
\]

\[
\text{fun} \ n => \text{case} \ n \ \text{of}
\]

\[
| 0 \Rightarrow \text{id}
\]

\[
| 1 \Rightarrow \text{id}
\]

\[
\text{let} \ n' => \text{box} (c, (q, q')) =>
\]

\[
(c, q) \leftarrow \text{gate} (\text{RGate} (2m-n')) (c, q);
\]

\[
\text{output} (c, (q, w))
\]

\[
\text{end}
\]

---

The presentation in this section is actually a simplification of the work of Krishnaswami et al. (2015), as we do not consider linear types with any dependencies.
The Fourier transform can now be defined in a type-safe way:

\[
\begin{align*}
\text{fourier : } & (n : \text{Nat}). \text{CIRC}(\otimes n \text{ qubit}, \otimes n \text{ qubit}) \Rightarrow \\
& \text{fun } n \Rightarrow \text{case } n \text{ of} \\
& | 0 \Rightarrow \text{id} \\
& | 1 \Rightarrow \text{hadamard} \\
& | S n' \Rightarrow \text{box} (q,\omega) \Rightarrow \\
& \text{unbox rotations} (S n') n' (q,\omega)
\end{align*}
\]

where \text{hadamard} = \text{box } w \Rightarrow \text{gate } H. w.

7. Discussion

Thus far we have shown that Q\text{WIRE} is a small, safe, and expressive circuit language. In the remainder of the paper we take a closer look at the similarities and differences between Q\text{WIRE} and existing quantum circuit languages, with an eye towards future work.

7.1 The QRAM model

The driving design of Q\text{WIRE} is the separation of classical computations in the host language from quantum computations in the circuit language. The inspiration for this model comes from two main sources.

On the logical side, Q\text{WIRE} draws on Benton’s (1995) linear/non-linear logic (LNL), which partitions the exponential from Girard’s linear logic (1987) into a purely linear fragment and a purely non-linear fragment connected via a categorical adjunction. Variations on LNL have extended the logical framework to type systems for other substructural logics (Pfenning and Griffith 2015), polarized logics (Zeilberger 2008), and dependently-typed logics as in Section 6.2 (Krishnaswami et al. 2015).

On the quantum computing side, the QRAM model postulates a classical computer working alongside a quantum computer. Q\text{RAM} is widely accepted as a programming model, although there is no clear consensus as to the degree to which the structure of quantum programming languages should reflect this separation.

At one end of the Q\text{RAM} spectrum of language design is Q\text{WIRE}, which syntactically separates quantum data inside circuits from classical data, and treats these two syntactic fragments as distinct languages. Betelli et al.’s Q programming language (2003), takes a similar approach, treating circuits (called quantum operators) as an isolated subsystem inside a generic host language.

Quipper and LIQ\text{U}/i) are based on the Quantum IO Monad (Altenkirch and Green 2010), which isolates quantum operations behind a monad. Indeed, the adjoint structure of Q\text{WIRE}, when viewed from the host language, forms a similar monad, where the bind of the monad is implemented with dynamic lifting. However, unlike in Q\text{WIRE}, qubits are first-class data in these systems, even though they cannot be constructed outside of the monad.

The separation between circuits and ordinary data has proved useful in the design of classical circuit languages as well. For example, in Haskell the \text{arrow} type class can be used to describe functional structures such as those corresponding to circuits (Hughes 2005). The fundamental constructor of arrows, which coerces a function in the host language into an arrow type, is not valid for Q\text{WIRE}, although arrows have applications for non-circuit models of quantum computation (Vizzotto et al. 2009).

On the opposite end of the spectrum are languages like QML (Altenkirch and Gratting 2005), the quantum \text{\lambda}-calculus (Selinger and Valiron 2009), and QPL (Selinger 2004), which avoid dealing with circuits entirely by treating qubits as data. Having first-class qubits may lead to more natural programming abstractions, like partially applied higher-order functions or imperative loops. However it requires a much more involved type theory (for instance, linear subtyping in the quantum \text{\lambda}-calculus) to achieve type safety.

7.2 Type systems for well-formed circuits

Q\text{WIRE} provides a type-safe circuit language within an arbitrary (type-safe) host language by keeping the circuit language minimal and pushing the remaining infrastructure to the host language. Embedded languages like Quipper and LIQ\text{U}/i) do not cleanly separate embedded circuits from the host language, which means that verifying the embedded language requires verifying the combination host and circuit languages. For Q\text{WIRE} we have shown that runtime errors in circuits can only arise from the host language, which is a maximal guarantee while still allowing arbitrary classical computations.

The type-safety guarantees gained from linear logic (e.g. respecting the no-cloning theorem) have been well-established by the quantum \text{\lambda}-calculus (Selinger and Valiron 2009). Quipper comes equipped with a programming idiom that recommends using quantum variables linearly except in certain circumstances, but programmers are unlikely to consistently follow this convention because it is neither enforced at compile time nor presented as a collection of unambiguous rules.

The Proto-Quipper project is an attempt to apply these foundations to a core language for Quipper with the goal of better runtime guarantees (Ross 2015). However, Proto-Quipper covers only a small subset of Quipper, and does not include measurement or initialization of qubits. Further, the classical component of Proto-Quipper is fixed, as it must be compatible with the underlying linear type system. Proto-Quipper is not a pure language, because its operational semantics imperatively constructs a circuit as the program runs and there is no equational theory. In contrast, the semantics of Q\text{WIRE} is pure and equational reasoning is valid. Finally, the type system of Proto-Quipper makes extensive use of subtyping to account for linear use of quantum data. Although the type system makes it easier to write code without linearity annotations, it makes it harder to know when a term is well-typed. In Q\text{WIRE}, the separation between the host language and circuit language makes linear typing easy and subtyping unnecessary.

An alternative to a linear type system is the Quantum IO Monad (Altenkirch and Green 2010). Although the monadic approach is sufficient to enforce no-cloning, by itself it is not strong...
enough to avoid all runtime errors. For example, Altenkirch and Green point out that extra semantic conditions based on the weakening property from linear logic are needed to safely type locally-bound ancilla and unitary conditional statements.

Although LIQW[i]’s type system is loosely based on the Quantum IO monad, in LIQU[i] qubits and circuits are dynamically typed, and so certain operations, such as circuit reversal, may fail at runtime. Furthermore, LIQU[i] gates can always be applied to a list of qubits with the intention of operating on only a finite prefix of them. If the list is empty, any such operation could fail.

### 7.3 Denotational semantics and formal verification

Proto-Quipper has a type-safe operational semantics, but not a denotational semantics against which to compare. Conversely, LIQU[i] has a built-in denotational semantics since entangled qubits are represented directly by their ket state, which allows for the formal analysis of algorithms.

Although formal verification of algorithms is time-consuming, in the case of quantum computing the cost is likely worthwhile: quantum computing resources will be expensive for the foreseeable future, debugging is doubly difficult in a quantum setting, and testing using simulations is not scalable. Verification efforts related to LIQU[i] include an efficient compiler for a reversible fragment of the language in the formal theorem prover P∗ (Amy et al. 2016). Other verification projects based on denotational semantics for a variety of quantum languages exist on paper but not as machine-checked proofs for various simple quantum programming languages (D’Hondt and Panangaden 2006; Kakutani 2009; Ying 2011).

We expect QWire to be amenable to a similar kind of verification based on the denotational semantics presented in Section 5. In particular, we are interested in using a dependently-typed theorem prover like Coq (Coq Development Team 2015) as a host language, and using it to prove theorems about circuits. In fact, the dependently-typed infrastructure described in Section 6.2 was inspired by our investigations into a Coq implementation.

Verification based on equational theories of quantum computation (Staton 2015) is also well-suited for QWire. These equational theories characterize the semantic equivalence of circuits, such as the fact that \(\text{in2Seq H H} = \text{equiv} \) is equivalent to the identity circuit. Such a theory could justify circuit optimizations provide a syntactic framework for program verification.

### 7.4 Usability

As a core circuit language, QWire is still missing many of the advanced features provided by Quipper and LIQU[i]. As we look towards implementations of QWire in various host languages, we can learn from the features of more mature languages.

**Parametric operators on circuits.** Quipper and LIQU[i] both provide operations that globally transform circuits, including circuit reversal replacing one universal gate set with another, and applying optimizations. In general these operations are built into the language, and may fail at runtime if various conditions are not met. In QWire we have already illustrated how these operations can be written directly in the host language by (safely) extending it with a case analysis operation on circuits.

**Automatic generation of quantum oracles.** Quipper’s quantum oracle feature uses Template Haskell (Sheard and Jones 2002) to generate a quantum circuit from an arbitrary classical function. By using Haskell as a host language we can imagine a similar extension to QWire.

**Scalability.** Quipper and LIQU[i] have both been used to successfully implement many nontrivial quantum algorithms (Siddiqui et al. 2014; Green et al. 2013a; Wecker and Svore 2014), in which the size of quantum circuits can grow into the millions of gates. One approach to scalability, embraced by LIQU[i], involves aggressive optimization and simulation, and is compatible with QWire using circuit case analysis. Another approach is to represent some circuits as black boxes when they are to be reused many times, recording their definition only once and (for example) precomputing their simulated behavior. This feature could be integrated into QWire by means of a function of type \(\text{Circ}(W_1, W_2) \rightarrow \text{Gate}(W_1, W_2)\) that coerces boxed circuits into host-level gates.

**Quantum data types.** A quantum data type is any data type consisting of qubits, which is useful for describing modules like the quantum integers. Quipper provides a typeclass-based approach to quantum data types consisting of a data type of qubits along with a corresponding classical data type of booleans (corresponding to the lifted type \(|W\rangle\) in the syntax of QWire). In this paper we only consider tuples, but an extended system could easily allow other finite data types. Infinite data types are more problematic—in Quipper, infinite data types like lists must be instantiated at a finite size before generating circuits for them. A better solution is to include finitely indexed data types, such as the n-ary tuples of qubits shown in Section 6.2. Instantiation is enforced by the fact that \(\text{II (x : Nat. Cir}(x \& x)\text{qubit) is not itself a circuit; it is a family of circuits that can be instantiated by feeding it a concrete natural number.}

### 7.5 Conclusion

QWire is a minimal and highly modular core circuit language. It is minimal in that QWire has only five distinct commands, two of which are eliminated in the normalization procedure. It is modular in that QWire isn’t attached to any specific programming language. We expect that the QWire interface will be useful in dependently-typed host languages like Coq for verification and formal analysis of circuits, in higher-order functional languages like Haskell, OCaml or F#, or potentially even in imperative languages like Python, Java, or C.

QWire uses linear types to enforce no-cloning, but does not allow them to spill over into the host language. This is crucial because linear types are the most natural way to enforce no-cloning, but are tremendously difficult to integrate into existing languages. QWire gets the best of both worlds by ensuring that circuits are linearly typed while allowing an arbitrarily powerful type system in the classical host language.

As a circuit description language, QWire is a low-level piece in the development of sophisticated quantum programming languages. Ultimately however, all quantum computation will boil down to circuit generation, necessitating the use of a circuit language like QWire. Having QWire as a safe, small circuit language is an excellent building block on which to rest the complex world of quantum computation.

### Acknowledgments

We are grateful to Peter Selinger for his insights into quantum programming languages. This work is supported in part by the ONR MURI No. FA9550-16-1-0082, and by NSF Grants No. CCF-1421193 and DGE-1321851.
References


