QWIRE: A QRAM-Inspired Quantum Circuit Language

Jennifer Paykin
jpaykin@seas.upenn.edu

Robert Rand
rrand@seas.upenn.edu

Steve Zdancewic
stevez@cis.upenn.edu

University of Pennsylvania

We introduce QWIRE, a linearly typed, strongly normalizing language for quantum circuits inspired by the QRAM model for quantum computing. We demonstrate the type-safety of the language and consider its embeddings in rich host languages, with the aim of producing an expressive and fully verified circuit representation language.

1 Introduction

The QRAM model for quantum computing [8] offers a powerful foundation for interacting with a quantum computer. In this model, a classical computer handles the majority of computing tasks with the help of a quantum random access machine for performing specialized quantum operations. These quantum operations are often packaged up into *circuits* in order to modularize code and minimize latency of communication between the classical and quantum machines. The QRAM model provides a useful separation for classical and quantum algorithms, and is thought to be the most feasible architecture for practical quantum computing [11, 16].

Several languages have been developed around the QRAM model. On the imperative side, these include the C-like QCL [12, 13] and Bettelli et al.'s C++ library [3]. On the functional side, they include the Altenkirch and Grattage's QML [1] and the Haskell embedded DSL Quipper [7, 6]. These languages focus on the generation of circuits using expressive, powerful, and complex programming paradigms.

In this paper we depart from these languages to develop a core theory of quantum circuits themselves. The goal is the syntactic representation of circuits with a safe type system and a sound equational semantics. Based on process calculi for classical linear logic [18] and on proof nets for quantum circuits [19], QWIRE is a small, safe, pure core for describing circuits. Leveraging this simple structure, in future work we anticipate verifying both quantum algorithms and the language itself in a theorem-prover like Coq [5], which could serve as both an implementation and a verification platform. We also imagine using QWIRE as a compilation target for other circuit generation languages.

However, QWIRE is not just a language of circuits. The strength of the embedded circuit-generation languages we mentioned comes from being able to use arbitrary classical computation to manipulate quantum circuits. This foundation of the QRAM model is at the forefront of QWIRE, which presents a simple but explicit interface between circuits and an arbitrary functional host language.

The choice to focus on functional host languages is based on lessons learned from previous quantum languages, including the Quantum Lambda Calculus [15], QML [1], and Quipper [7]. Typed functional programming languages have particular appeal for writing quantum programs due to their reliability and suitability for mathematical analysis. Generally speaking, if a well-typed functional program compiles, the static type-checker guarantees that it is free of runtime errors. This property is particularly valuable in the case of a quantum computing device for which failure might be very expensive. Moreover, the type discipline of functional programs makes them more amenable to mathematical modeling[1], which in turn

[1] For instance, functional quantum programs can structured monadically [2].

Submitted to:
QPL 2016

means that that it is easier to formally verify the correctness of such programs. Full program verification is expensive, but in the case of quantum computers, where the cost of quantum-computing resources is likely to be large, such verification efforts are warranted.

Contributions. Taking inspiration from the QRAM model, we propose a type-safe quantum programming language that is amenable to mathematical analysis. We present a small circuit language, QWIRE, that is both strongly normalizing and type-safe, with linear types that respect the laws of quantum mechanics. We have designed QWIRE with the intention that, when embedded in an appropriate host language, it can express sophisticated quantum algorithms, serve as a compilation target for Quipper or another circuit generation language, and still be suitable for formal verification.

We introduce the syntax and type system of QWIRE in Section 2, and give examples of some simple circuits. Section 3 gives a precise operational semantics to the language and establishes type safety and normalization results. In Section 4 we consider how richer host-language features, such as the ability to analyze circuits or the presence of dependent types, can be effectively used for quantum programming. Finally, in Section 5 we compare QWIRE more thoroughly to an existing formalization of Quipper, highlighting the advantages of each system. We conclude with a look at the next steps towards building a highly expressive and rigorously checked platform for quantum computing.

2 The QWIRE Circuit Language

A circuit can be interpreted in two equally valid ways. On the one hand, a circuit can be thought of as a relation between sets of wires. A rewiring circuit, such as $(w_1, w_2) \leftrightarrow (w_3, ())$, depicts such a relation. On the other hand, a circuit can be thought of as a function from one wire type to another. We write

$$\mathsf{gate}\ w' = \mathsf{H}\ w\ \mathsf{in}\ C$$

to depict the *application* of a Hadamard gate to the wire w, binding the output to the wire name w' in the circuit C. This functional interpretation allows for higher-order operations on circuits and provides a comfortable programming environment for functional programmers. QWIRE attempts to integrate the best of these two perspectives by using a functional syntax to represent relational operations.

A circuit is paramaterized by the names and types of its input and output wires. The types of wires include qubits, bits, units, and products of wires. Wire contexts Ω are collections of wire names w with their corresponding wire type.

$$W ::= \mathsf{qubit} \mid \mathsf{bit} \mid 1 \mid W_1 \otimes W_2$$
$$\Omega ::= \cdot \mid w : W \mid \Omega_1, \Omega_2$$

The typing judgment of a circuit has the form $\Gamma; \Omega_1 \vdash C \mid \Omega_2$ where Γ is a collection of residual classical variables (whose use is explained below), Ω_1 denotes the input wires to C, and Ω_2 denotes the output wires of C.

In the rest of this section we will introduce the four ways of constructing circuits: rewirings and compositions, which give a relational interface with circuits, and gate applications and unboxing, which present a functional view.

Rewiring. A simple rewiring circuit assigns a new name to a wire.

$$\overline{\Gamma; w_1 : W \vdash w_1 \leftrightarrow w_2 \mid w_2 : W}$$

More complex rewirings package up a pattern of wires to another pattern, giving information about how to reassociate multiple wires at once. For example, the circuit $w \leftrightarrow (w_1, w_2)$ breaks up an input wire w of type $W_1 \otimes W_2$ into two output wires $w_1 : W_1$ and $w_2 : W_2$. As another example, the circuit $() \leftrightarrow w$ has no inputs and a single output wire w of type 1.

Rewiring is generalized by a theory of *wire patterns*, denoted p. Patterns consist of unit patterns, tuples of patterns, and wires themselves. In addition, the pattern $\mathsf{Q}\,w$ matches only a wire of type qubit, while the pattern $\mathsf{B}\,w$ matches only a wire of type bit. The plain wire name pattern w matches a wire of any type.

$$p ::= w \mid \mathsf{Q}\,w \mid \mathsf{B}\,w \mid () \mid (p_1, p_2)$$

A pattern can be thought of as a description of how to associate one collection of wires in a context into a single wire type. This judgment is denoted $\Omega \vdash p : W$, and is defined as follows:

$$\overline{w : W \vdash w : W} \qquad \overline{w : \mathsf{qubit} \vdash \mathsf{Q}\,w : \mathsf{qubit}} \qquad \overline{w : \mathsf{bit} \vdash \mathsf{B}\,w : \mathsf{bit}}$$

$$\overline{\cdot \vdash () : 1} \qquad \frac{\Omega_1 \vdash p_1 : W_1 \quad \Omega_2 \vdash p_2 : W_2}{\Omega_1, \Omega_2 \vdash (p_1, p_2) : W_1 \otimes W_2}$$

Note that here and throughout the paper we use Ω_1, Ω_2 to denote the *linear* combination of the wire contexts—we implicitly require that Ω_1 and Ω_2 bind *disjoint* sets of wire names. The rules also ensure that wires are never dropped. These properties are crucial for ensuring both that circuits are well-formed and that quantum data is not used by the circuit in a way that violates the no-cloning theorem.

A general rewiring circuit describes how to pack and unpack a collection of wires into a different collection by moving in and out of a single wire type:

$$\frac{\Omega_1 \vdash p_1 : W \quad \Omega_2 \vdash p_2 : W}{\Gamma; \Omega_1 \vdash p_1 \leftrightarrow p_2 \mid \Omega_2}$$

Gates. The next type of circuit is gate application. Gates can be thought of as opaque operators on wires. Unitary gates, denoted u, are derived from some standard unitary gate set plus conjugate transpose and qubit and bit controls. Non-unitary gates are denoted g, and include ways to construct new wires, measure qubits, and discard and duplicate classical bit wires.

$$u ::= \mathsf{H} \mid \mathsf{QNOT} \mid \mathsf{C}\,u \mid \mathsf{CC}\,u \mid u^\dagger \mid \cdots$$
$$g ::= u \mid \mathsf{qinit}_0 \mid \mathsf{qinit}_1 \mid \mathsf{cinit}_0 \mid \mathsf{cinit}_1 \mid \mathsf{meas} \mid \mathsf{cdiscard} \mid \mathsf{cdup}$$

The input and output types of a gate are given by a pair of functions Σ_{In} and Σ_{Out}:

u	$\Sigma_{\mathsf{In}}(u)$	$\Sigma_{\mathsf{Out}}(u)$
H	qubit	qubit
QNOT	qubit	qubit
$u^\dagger$	$\Sigma_{\mathsf{Out}}(u)$	$\Sigma_{\mathsf{In}}(u)$
$\mathsf{C}\,u$	$\mathsf{qubit} \otimes \Sigma_{\mathsf{In}}(u)$	$\mathsf{qubit} \otimes \Sigma_{\mathsf{Out}}(u)$
$\mathsf{CC}\,u$	$\mathsf{bit} \otimes \Sigma_{\mathsf{In}}(u)$	$\mathsf{bit} \otimes \Sigma_{\mathsf{Out}}(u)$

g	$\Sigma_{\mathsf{In}}(g)$	$\Sigma_{\mathsf{Out}}(g)$
$\mathsf{qinit}_0, \mathsf{qinit}_1$	1	qubit
$\mathsf{cinit}_0, \mathsf{cinit}_1$	1	bit
meas	qubit	bit
cdiscard	bit	1
cdup	bit	$\mathsf{bit} \otimes \mathsf{bit}$

We write $(W_1, g, W_2) \in \Sigma$ to mean that $\Sigma_{\mathsf{In}}(g) = W_1$ and $\Sigma_{\mathsf{Out}}(g) = W_2$.

To construct a circuit using a gate, QWIRE provides a convenient let-style notation to indicate that a gate should be applied to a particular wire w_1. The output of the gate is then bound to w_2 and is used in the remainder of the circuit.

$$\frac{\Gamma; \Omega_0, w_2 : \Sigma_{\mathsf{Out}}(g) \vdash C \mid \Omega}{\Gamma; \Omega_0, w_1 : \Sigma_{\mathsf{In}}(g) \vdash \mathsf{gate}\ w_2 = g\ w_1\ \mathsf{in}\ C \mid \Omega}$$

As in the rewiring circuit, we can generalize gate application to arbitrary patterns of wires. The general form of the typing rule packs up the input wires and unpacks the output wires in a single statement:

$$\frac{\Omega_1 \vdash p_1 : \Sigma_{\mathsf{In}}(g) \quad \Omega_2 \vdash p_2 : \Sigma_{\mathsf{Out}}(g) \quad \Gamma; \Omega_0, \Omega_2 \vdash C \mid \Omega}{\Gamma; \Omega_0, \Omega_1 \vdash \mathsf{gate}\ p_2 = g\ p_1\ \mathsf{in}\ C \mid \Omega}$$

Composition. The composition of two circuits is written $\langle C_1 \mapsto C_2 \rangle$ and it connects the output wires of C_1 to the input wires of C_2 whenever the wires have the same name.

$$\frac{\Gamma; \Omega_1 \vdash C_1 \mid \Omega_1', \Omega \quad \Gamma; \Omega_2, \Omega \vdash C_2 \mid \Omega_2'}{\Gamma; \Omega_1, \Omega_2 \vdash \langle C_1 \mapsto C_2 \rangle \mid \Omega_1', \Omega_2'}$$

2.1 Circuits in the Classical World.

QWIRE is intended to be embedded in a traditional programming language, which we call the host language. In general, this should be a full functional programming language, complete with higher-order functions, abstract data types, recursion, etc. For the purposes of this presentation we use the simply-typed λ-calculus with booleans, although examples may make use of other simple data types.

We start by extending the host language with the Quipper-inspired type $\mathsf{CIRC}(W_1, W_2)$ to represent a closed, completed circuit in a functional style. To construct a term of type $\mathsf{CIRC}(W_1, W_2)$ from a concrete circuit C, we use a *box* to package up the inputs and outputs into single input patterns and output patterns. The terms and types of the simply-typed host language can be summarized as follows:

$$\begin{aligned} A &::= \mathsf{Bool} \mid A_1 \to A_2 \mid \mathsf{CIRC}(W_1, W_2) \\ t &:= x \mid \mathsf{true} \mid \mathsf{false} \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3 \mid \lambda x.t \mid t_1\, t_2 \mid \mathsf{box}\ (p_1 \Rightarrow C)\ \mathsf{return}\ p_2 \end{aligned}$$

The typing rules for this basic host language are unsurprising. Typing contexts Γ are collections of variables with types, and typing judgments have the form $\Gamma \vdash t : A$. The typing rule for $\mathsf{CIRC}(W_1, W_2)$ is necessarily more complex:

$$\frac{\Omega_1 \vdash p_1 : W_1 \quad \Gamma; \Omega_1 \vdash C \mid \Omega_2 \quad \Omega_2 \vdash p_2 : W_2}{\Gamma \vdash \mathsf{box}\ (p_1 \Rightarrow C)\ \mathsf{return}\ p_2 : \mathsf{CIRC}(W_1, W_2)}$$

This term packages up a circuit by binding the input wires given by the pattern p_1 and collecting together the output wires given by p_2.

Significantly, a term of type $\mathsf{CIRC}(W_1, W_2)$ can be used as if it were a function on wires inside another circuit.

$$\frac{\Omega_1 \vdash p_1 : W_1 \quad \Gamma \vdash t : \mathsf{CIRC}(W_1, W_2) \quad \Omega_2 \vdash p_2 : W_2 \quad \Gamma; \Omega_0, \Omega_2 \vdash C \mid \Omega}{\Gamma; \Omega_0, \Omega_1 \vdash \mathsf{unbox}\ p_2 = t\ p_1\ \mathsf{in}\ C \mid \Omega}$$

Figure 4 in Appendix A summarizes the circuit and host languages with their typing rules.

```
inSeq :: CIRC(W1,W2) -> CIRC(W2,W3) ->
         CIRC(W1,W3)
inSeq x y = box (w1 =>
    unbox w2 = x w1 in
    unbox w3 = y w2 in
    id w3)
  return w3
```

```
inPar :: CIRC(W1,W2) -> CIRC(W1',W2') ->
         CIRC(W1⊗W1', W2⊗W2')
inPar x y = box ((w1,w1') =>
    unbox w2  = x w1  in
    unbox w2' = y w1' in
    id (w2,w2'))
  return (w2,w2')
```

Figure 1: Parallel and sequential composition functions on term-level circuits

```
bell00 :: CIRC(1,qubit⊗qubit)
bell00 = box (() =>
    gate a = qinit0 () in
    gate a = H a in
    gate b = qinit0 () in
    gate (a,b) = (C QNOT) (a, b) in
    id (a,b))
  return (a,b)
```

```
alice : CIRC(qubit⊗qubit, bit⊗bit)
alice = box ((q,a) =>
    gate (q,a) = (C QNOT) (q,a) in
    gate q     = H q in
    gate x     = MEAS q in
    gate y     = MEAS a in
    id (x,y))
  return (x,y)
```

```
bob : CIRC(qubit⊗(bit⊗bit), qubit)
bob = box ((b,(x,y)) =>
    gate (y,b) = (CC X) (y,b) in
    gate (x,b) = (CC Z) (x,b) in
    gate ()    = cdiscard x  in
    gate ()    = cdiscard y  in
    id b)
  return b
```

```
teleport :: CIRC(qubit,qubit)
teleport = box(q =>
    unbox (a,b) = bell00 () in
    unbox (x,y) = alice (q,a) in
    unbox b     = bob (b,(x,y)) in
    id b)
  return b
```

Figure 2: A $\mathcal{Q}$WIRE implementation of quantum teleportation.

2.2 Examples

We can now show some example circuits written in $\mathcal{Q}$WIRE. For convenience, we'll begin by defining $\mathsf{id}(p)$ as a functional abbreviation for $p \leftrightarrow p$; it will frequently be used to terminate a circuit.

We start with some simple programs for manipulating term-level representations of circuits. The identity circuit $\mathtt{ID} : \mathsf{CIRC}(W,W)$ is defined as $\mathsf{box}\ (w \Rightarrow \mathsf{id}(w))\ \mathsf{return}\ w$. We can also combine two circuits in sequence or in parallel as shown in Figure 1.

We borrow a more interesting example from Green et al.'s introduction to the Quipper language [6]. Figure 2 shows the quantum teleportation circuit, broken up into four parts. Alice is trying to send a qubit `q`, the input to the `teleport` circuit, to Bob. The circuit `bell00` constructs a fresh Bell pair of entangled qubits `a` and `b`, which are sent to Alice and Bob, respectively. Alice entangles and measures `a` and `q` and outputs a pair of classical bits `x` and `y`. Bob then uses these classical bits to transform his own qubit into the state of the original `q`.

Note that this example has been translated almost directly from Quipper, which after all was designed as a circuit generation language.

3 Operational semantics: circuit normalization

The ability to box, unbox, and compose circuits gives a good amount of expressivity to $\mathcal{Q}$WIRE. However, they somewhat obscure the structure of the circuit being constructed. In this section we define a circuit

normalization procedure that reduces all circuits to a small set of normal forms, defined as follows:

$$N ::= n_1 \leftrightarrow p_2 \mid \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ N$$
$$n ::= \mathsf{Q}\,w \mid \mathsf{B}\,w \mid () \mid (n_1, n_2)$$

Normal circuits are defined entirely by normal pattern rewiring and gate application, where normal patterns are patterns whose types are fully defined by their structure. In particular, we do not allow input wires of arbitrary type to be referenced in normal circuits (although output wires can be arbitrary).

This is a noteworthy result, because it means that all circuits can be characterized exactly by how their gates are applied. Proto-Quipper uses a similar process to build up partial circuits through its operational semantics, but in QWIRE circuits are syntactic objects, and the reduction rules are self-contained.

Only simple inputs! The normalization procedure works by simplifying all structural occurrences of patterns so that they appear only in gate applications and rewiring links. Suspiciously though, structural rules sometimes seem necessary, for instance when a circuit has a single input wire of type $\mathsf{qubit} \otimes \mathsf{qubit}$ but applies a gate to only one of the components as in the example below:

$$\cdot; w : \mathsf{qubit} \otimes \mathsf{qubit} \vdash \langle w \leftrightarrow (w_0, w_1) \mapsto \mathsf{gate}\ w_2 = \mathsf{H}\ w_1\ \mathsf{in}\ w_2 \leftrightarrow w_3 \rangle \mid w_0 : \mathsf{qubit}, w_3 : \mathsf{qubit}$$

There is no way to further reduce this circuit because its input wire cannot be decomposed using only normal circuits. In some sense, the above example is equivalent to a circuit with two qubit inputs, w_0 and w_1, and yet, on the other hand, reduction rules should preserve typing judgments.

Similarly to the λ-calculus, we claim that a circuit is only reducible when it its typing judgment has a particular form. In the λ-calculus we restrict the normalization procedure to closed terms, but in the circuit language we restrict it to circuits whose input has only simple wire types. That is, the input wires of a reducible circuit must be of the form $\mathcal{Q}$, where

$$\mathcal{Q} ::= \cdot \mid w : \mathsf{qubit} \mid w : \mathsf{bit} \mid \mathcal{Q}_1, \mathcal{Q}_2$$

Notice that if $\Omega \vdash n : W$ then Ω is a context of simple wire types. In addition, if $\mathcal{Q} \vdash p : W$ then there is a unique normal pattern n such that $\mathcal{Q} \vdash n : W$.

With this in mind, we introduce the normalization procedure for circuits.

Rewiring. If $\cdot; \mathcal{Q} \vdash p_1 \leftrightarrow p_2 \mid \Omega$, then there is some W such that $\mathcal{Q} \vdash p_1 : W$. As we just claimed, this means there is some unique normal pattern n_1 such that $\mathcal{Q} \vdash n_1 : W$. Thus the rewiring $p_1 \leftrightarrow p_2$ is equivalent to the normal circuit $n_1 \leftrightarrow p_2$.

$$\frac{\mathcal{Q}_1 \vdash n_1 : W}{p_1 \leftrightarrow p_2 : W \Longrightarrow n_1 \leftrightarrow p_2}$$

Notice that this rule is type directed, like an η-rule in the λ-calculus.

Gate Application. Since a gate application circuit is a normal form, we reduce structurally under it:

$$\frac{C \Longrightarrow C'}{\mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ C \Longrightarrow \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ C'}$$

This structural rule gets stuck if the output pattern of the gate is not normal. As in the rewiring η-rule, we can always find an α-equivalent circuit that uses only simple patterns.

$$\frac{\mathcal{Q}_1 \vdash n_1 : \Sigma_{\mathsf{In}}(g) \quad \mathcal{Q}_2 \vdash n_2 : \Sigma_{\mathsf{Out}}(g)}{\mathsf{gate}\ p_2 = g\ p_1\ \mathsf{in}\ C \Longrightarrow \mathsf{gate}\ n_2 = g\ p_1\ \mathsf{in}\ \langle n_2 \leftrightarrow p_2 \mapsto C \rangle}$$

Boxing and Unboxing. In order to unbox a term we first reduce it to weak-head normal form.

$$\frac{t \Longrightarrow t'}{\mathsf{unbox}\ p_2 = t\ p_1\ \mathsf{in}\ C \Longrightarrow \mathsf{unbox}\ p_2 = t'\ p_1\ \mathsf{in}\ C}$$

The β-rule for unboxing such a value extracts the internal circuit and uses rewirings to make the input and output wires match.

$$\frac{}{\mathsf{unbox}\ p_2 = (\mathsf{box}\ (p_1' \Rightarrow C_1)\ \mathsf{return}\ p_2')\ p_1\ \mathsf{in}\ C_2 \Longrightarrow \langle \langle p_1 \leftrightarrow p_1' \mapsto \langle C_1 \mapsto p_2' \leftrightarrow p_2 \rangle \rangle \mapsto C_2 \rangle}$$

From the perspective of the host language, a boxed circuit can normalize further, to a value of the form

$$\mathsf{box}\ (n_1 \Rightarrow N)\ \mathsf{return}\ n_2$$

where both the input and the output of the circuit are normal. Notice that circuit-level objects do not require the output wires to have simple types, only the input wires.

$$\frac{C \Longrightarrow C'}{\mathsf{box}\ (n_1 \Rightarrow C)\ \mathsf{return}\ n_2 \Longrightarrow \mathsf{box}\ (n_1 \Rightarrow C')\ \mathsf{return}\ n_2}$$

A boxed circuit may not necessarily be defined over normal patterns, however, but just like gate rewirings, it is equivalent to one that is. This fact is represented by the following η-rule, which generates fresh normal patterns to type the input and output of the circuit.

$$\frac{\mathcal{Q}_1 \vdash n_1 : W_1 \quad \mathcal{Q}_2 \vdash n_2 : W_2}{t : \mathsf{CIRC}(W_1, W_2) \Longrightarrow_\eta \mathsf{box}\ (n_1 \Rightarrow \mathsf{unbox}\ w = t\ n_1\ \mathsf{in}\ w \leftrightarrow n_2)\ \mathsf{return}\ n_2}$$

The η and structural reduction rules should only be applied at the top level, and not under an unbox operator.

Composition. The main challenge of the operational semantics is to eliminate circuit composition. The semantics first reduces the left circuit half of the composition to a normal form.

$$\frac{C_1 \Longrightarrow C_1'}{\langle C_1 \mapsto C_2 \rangle \Longrightarrow \langle C_1' \mapsto C_2 \rangle}$$

When the first component of the composition is a gate application, we can permute it to the front of the entire circuit.

$$\frac{}{\langle \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ N_1 \mapsto C_2 \rangle \Longrightarrow \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ \langle N_1 \mapsto C_2 \rangle}$$

After multiple applications of this rule, we are left with a rewiring circuit $n_1 \leftrightarrow p_2$ in the place of N_1. However, before we can reduce C_2 to a normal form, we must ensure the input of C_2 is simple. We thus proceed in two steps.

1) Parts of p_2 occur free in C_1. If some part of the rewiring's output occurs free in C_2 then our goal will be to substitute the pattern n_1 for the pattern p_2 in C_2. If p_2 is not just a wire name, it must be a tuple of patterns, and we can deconstruct it as follows:

$$\frac{\text{output}((n_1', n_2') \leftrightarrow (p_1', p_2')) \not\perp \text{input}(C)}{\langle (n_1', n_2') \leftrightarrow (p_1', p_2') \mapsto C \rangle \Longrightarrow \langle n_1' \leftrightarrow p_1' \mapsto \langle n_2' \leftrightarrow p_2' \mapsto C \rangle \rangle}$$

Here we write $X_1 \not\perp X_2$ to mean that names sets X_1 and X_2 intersect; below we use $X_1 \perp X_2$ to mean that they are disjoint. Since the input of the normal rewiring is simple, we will never have a mismatched rewiring.

Once the output of the rewiring is a single wire, we can perform a substitution of the input pattern for the wire.

$$\frac{w_2 \in \text{input}(C)}{\langle n_1 \leftrightarrow w_2 \mapsto C \rangle \Longrightarrow C\{n_1/w_2\}} \qquad \frac{w_2 \in \text{input}(C)}{\langle \mathsf{Q}\, w_1 \leftrightarrow \mathsf{Q}\, w_2 \mapsto C \rangle \Longrightarrow C\{w_1/w_2\}} \qquad \frac{w_2 \in \text{input}(C)}{\langle \mathsf{B}\, w_1 \leftrightarrow \mathsf{B}\, w_2 \mapsto C \rangle \Longrightarrow C\{w_1/w_2\}}$$

Here $C\{p/w\}$ is the usual notion of capture-avoiding substitution. Since each wire occurs exactly once in C, the resulting circuit is guaranteed to be well-typed.

2) p_2 is disjoint from C_1. On the other hand, if the output of the rewiring does *not* occur in the input of C_2, then the input to C_2 is simply typed and we can safely reduce the right half of the circuit.

$$\frac{\text{output}(n_1 \leftrightarrow p_2) \perp \text{input}(C_2) \quad C_2 \Longrightarrow C_2'}{\langle n_1 \leftrightarrow p_2 \mapsto C_2 \rangle \Longrightarrow \langle n_1 \leftrightarrow p_2 \mapsto C_2' \rangle}$$

Once C_2 reaches a normal form N_2, we can permute gate applications in N_2 to the front of the circuit.

$$\frac{\text{output}(n_1 \leftrightarrow p_2) \perp \text{input}(\mathsf{gate}\ n_2' = g\ n_1'\ \mathsf{in}\ C_2)}{\langle n_1 \leftrightarrow p_2 \mapsto \mathsf{gate}\ n_2' = g\ n_1'\ \mathsf{in}\ C_2 \rangle \Longrightarrow \mathsf{gate}\ n_2' = g\ n_1'\ \mathsf{in}\ \langle n_1 \leftrightarrow p_2 \mapsto C_2 \rangle}$$

Finally, when the two halves of the composition are rewiring circuits that don't share any wires between them, we can construct a new rewiring circuit that performs the two existing rewirings in parallel.

$$\frac{\text{output}(n_1 \leftrightarrow p_2) \perp \text{input}(n_1' \leftrightarrow p_2')}{\langle n_1 \leftrightarrow p_2 \mapsto n_1' \leftrightarrow p_2' \rangle \Longrightarrow (n_1, n_1') \leftrightarrow (p_2, p_2')}$$

The complete operational semantics is summarized in Figure 5 of Appendix A.

Type safety. We prove type safety with progress and preservation theorems, which are mutually recursive between terms and circuits.

Theorem 1 (Preservation). *If $\vdash t : A$ and $t \Longrightarrow t'$ then $\vdash t' : A$. If $\cdot; \mathcal{Q} \vdash C \mid \Omega$ and $C \Longrightarrow C'$ then $\cdot; \mathcal{Q} \vdash C' \mid \Omega$.*

Theorem 2 (Progress). *If $\vdash t : A$ then t is a value or t can take a step. If $\cdot; \mathcal{Q} \vdash C \mid \Omega$ then C is normal or C can take a step.*

We also prove that circuits are normalizing, provided that the term language is also normalizing.

Theorem 3 (Normalization). *If $\cdot \vdash t : A$ then there exists some value v such that $t \Longrightarrow^* v$. If $\cdot; \mathcal{Q} \vdash C \mid \Omega$ then there exists some N such that $C \Longrightarrow^* N$.*

Due to space constraints, we leave the (mostly standard) proofs of these theorems for an expanded report.

4 QWIRE in a rich host language

4.1 A meta-level case analysis of circuits

Thanks to the operational semantics, we know that every host-language circuit $t : \mathsf{CIRC}(W_1, W_2)$ evaluates to a representation of a normal circuit with a very simple structure. This leads to a powerful observation: it is feasible to perform case analysis on circuits in the host language. As a consequence, programmers can write sophisticated analyses of circuits in the host language itself.

The quintessential example of such circuit manipulation is a safe circuit-reversal function.

$$\texttt{reverse} :: \mathsf{CIRC}(W_1, W_2) \to \mathsf{Maybe}\,\mathsf{CIRC}(W_2, W_1)$$

A simple implementation of this function simply checks whether all gates in the given circuit are unitary, and, if so, reverses them and their order:

```
reverse : CIRC(W1,W2) -> Maybe CIRC(W2,W1)
reverse (Rewire x)   = Some (toCirc (toGate (transpose x)))
reverse (Gate x g z) =
        case (toUnitary g, reverse z) of
        | (Some u, Some z') ->
          let y' = inPar ID (toCirc (toGate (transpose u))) in
          let x' = toCirc (sym x) in
          Some (inSeq z' (inSeq y' x'))
        | (_, _) -> None
        end
```

The reverse example considers the two possible normal forms of the input circuit. In the first case the input is a rewiring circuit between the wire types W_1 and W_2. This host-level rewiring is represented by an isomorphism between the two wire types. To represent this relationship in the host language we add a type of wire isomorphisms.

$$A ::= \cdots \mid \mathsf{ISO}(W_1, W_2)$$

We leave the details of the implementation of wire isomorphisms to future work, but we specify an interface for them in Figure 3. The type of isomorphisms should certainly be an equivalence relation, and we expect to be able to coerce an isomorphism back into a concrete circuit.

In the second branch of the reverse example, g is the first gate applied in the circuit. The gate is only applied to a section of the input; assume its signature is (W_1', g, W_2'). The remainder of the circuit, which is bound to the variable `z` in the example, uses the output of g, which means it has the type $\mathsf{CIRC}(W_0 \otimes W_2', W_2)$, where W_0 is the remainder of W_1 when you take out the input to the gate, W_1'. The entire circuit t can be represented pictorially as follows:

$$W_1 \cong W_0 \otimes W_1' \xrightarrow{g} W_0 \otimes W_1' \xrightarrow{N} W_2$$

The isomorphism in the first step is represented by the variable x of type $\mathsf{ISO}(W_1, W_0 \otimes W_1')$.

The `reverse` example also assumes a host-level representation of gates, which includes the interface shown in Figure 3.

$$A ::= \cdots \mid \mathsf{GATE}(W_1, W_2) \mid \mathsf{UNI}(W_1, W_2)$$

$$\texttt{refl} : \mathsf{ISO}(W, W)$$
$$\texttt{sym} : \mathsf{ISO}(W_1, W_2) \to \mathsf{ISO}(W_2, W_1)$$
$$\texttt{trans} : \mathsf{ISO}(W_1, W_2) \to \mathsf{ISO}(W_2, W_3) \to \mathsf{ISO}(W_1, W_3)$$
$$\texttt{toCirc} : \mathsf{ISO}(W_1, W_2) \to \mathsf{CIRC}(W_1, W_2)$$

$$\texttt{toGate} : \mathsf{UNI}(W_1, W_2) \to \mathsf{GATE}(W_1, W_2)$$
$$\texttt{toUni} : \mathsf{GATE}(W_1, W_2) \to \mathsf{Maybe}\,\mathsf{UNI}(W_1, W_2)$$
$$\texttt{toCirc} : \mathsf{GATE}(W_1, W_2) \to \mathsf{CIRC}(W_1, W_2)$$

Figure 3: Term-level interface for wire isomorphisms, gates, and unitary gates.

4.2 Dependent wire types

Another way to gain expressivity of circuits is by adding dependent wire types. Although linear dependent types are still an area of active research, the syntactic separation of the linear and non-linear types in QWIRE gives us an avenue for adding wire types that depend only on host terms, in the style of Krishnaswami [9]. For example, the dependent types `Bits n` and `Qubits n` encode length-indexed lists:

```
bits :: Nat -> WType                    qubits :: Nat -> WType
bits Zero        = 1                    qubits Zero        = 1
bits (Succ Zero) = bit                  qubits (Succ Zero) = qubit
bits (Succ n')   = bit ⊗ (Bits n')      qubits (Succ n')   = qubit ⊗ (qubits n')
```

In the following example we use these length-indexed lists to write a dependently-typed quantum Fourier transform in the style of the Quipper introduction [6]. 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 controlled R gates [6] as boxed circuits. Note that `rotations` takes two natural number inputs: m, representing the argument given to the controlled R Gates, and n, which indexes the number of bits in the input.

```
rotations :: forall (m n : Nat), CIRC(Qubits n, Qubits n)
rotations m Zero        = ID
rotations m (Succ Zero) = ID
rotations m (Succ n')   =  box ( (c,(q,w)) =>
                              unbox (c,w) = (rotations m n') (c,w) in
                              unbox (c,q) = (CRGate (2 + m - n')) (c,q) in
                              id (c,(q,w)) )
                            return (c,(q,w))
```

The Fourier transform can now be defined in a type-safe way.

```
fourier :: forall (n : Nat), CIRC(Qubits n, Qubits n)
fourier Zero        = ID
fourier (Succ Zero) = HADAMARD
fourier (Succ n')   = box ( (q,w) =>
                          unbox w = (fourier n') w in
                          unbox (q,w) = (rotations n' (Succ n')) (q,w) in
                          id (q,w) )
                        return (q,w)
```

5 Discussion

QWIRE is one of many languages based on the QRAM model. It's primary influence is Quipper [7, 6], arguably the state-of-the-art for typed, functional quantum programming. As a domain specific language

embedded in Haskell, Quipper takes full advantage of its host language, using higher-order types, type classes, and laziness to easily express circuits. On the other hand, Haskell cannot check that Quipper circuits are well-formed, in terms of linear use of quantum variables or distinguishing a reversible from an irreversible circuit, so it falls to the programmer to ensure that the circuits produced obey standard laws of quantum mechanics, like the no-cloning theorem. Furthermore, the semantics of Quipper is not well-suited to formal verification, meaning that bugs in the Quipper implementation itself could result in ill-formed circuits.

The Proto-Quipper project is an ongoing effort that seeks to overcome both of these limitations. As described by Ross [14], Proto-Quipper is a small, well-typed fragment of Quipper with a clearly defined operational semantics and a linear type system that guarantees compliance with the no-cloning theorem. In its current form, Proto-Quipper can express only a small fragment of Quipper programs, although work on Proto-Quipper is ongoing. Having introduced $\mathcal{Q}$WIRE and some of its possible extensions, it is worth considering what we've gained over the existing Proto-Quipper language, as well as what we've lost.

First-class qubits. Much of Proto-Quipper's expressive power comes from its ability to treat qubits and linear function spaces as first-class objects. For example, the teleport program from Section 2.2 could be restructured in Proto-Quipper as a closure, producing two entangled linear functions.

```
teleport :: () -> (qubit -> bit⊗bit) ⊗ (bit⊗bit -> qubit)
teleport () =
    let (a,b) = unbox bell00 in
    let f = curry (unbox alice) a in
    let g = curry (unbox bob) b in
    (f,g)
```

This example (courtesy of Peter Selinger) cannot be expressed in $\mathcal{Q}$WIRE, because the result is neither a circuit nor a term in the host language. Of course, a quantum computation must ultimately take the form of a closed circuit in order to be sent to the QRAM, hence `alice` and `bob` must eventually be joined (linearly) into a single circuit. Still, this modularity may be convenient, and its absence costs us a useful abstraction.

First-class circuits. Proto-Quipper is not based on one particular theory of circuits, but instead uses axiomatized circuit constructors for composition and reversal. Moreover, Proto-Quipper is not a pure language, because its operational semantics imperatively constructs a circuit as the program runs. This lack of purity also makes it harder to reason about a program's correctness because there is no overarching equational theory. In contrast, the semantics of $\mathcal{Q}$WIRE is pure, and equational reasoning is valid.

Having a concrete representation of circuits in $\mathcal{Q}$WIRE allows us to do meta-reasoning about them and explicit manipulation of them in the host language, as we saw in Section 4.1. Since Proto-Quipper doesn't have a concise normal form for it circuits, it doesn't admit these capabilities.

Linear type systems. The primary design principle in $\mathcal{Q}$WIRE is the separation of the host language from the linear circuit language. In Proto-Quipper the linear and non-linear fragments of the language are merged using the ! operator of linear logic. This makes it difficult to reason about advanced programming languages features, including polymorphism and dependent types, without getting bogged down by the linear type restrictions. In particular, the best approaches to integrating linear and dependent types is still an open problem [9, 10]. As we saw in Section 4.2, the modularity in $\mathcal{Q}$WIRE should allow us to add dependent types to the host language in a straightforward way.

Proto-Quipper's type system makes extensive use of subtyping, including the rule $!A \leq A$, which blurs the distinction between non-linear and linear types. This makes it easy to write programs with fewer type annotations, which is useful for programming. However, it comes with an additional cost to formalization and implementation—it is not immediately clear which parts of a program behave quantum-mechanically, and which parts behave classically. As a consequence, Proto-Quipper can express terms that are not quite circuits but are not quite classical, as in the teleport example above. QWIRE eliminates this ambiguity—expressions are either linear circuits, or regular, non-linear terms.

5.1 Future Work

In this paper we have begun the exploration of QWIRE as a core language for quantum circuits, but it is still very much a work in progress. We conclude by briefly outlining our plans for future work.

Host integration. With case analysis of circuits in the host language, the possibilities of host-language analysis of circuits expand beyond a simple reversal function. Some examples for possible future work include: (1) circuit optimization by collapsing the application of one unitary gate followed by its transpose; (2) transforming a circuit over one unitary gate set to another; (3) building alternative representations of circuits, like proof nets [19], for the purposes of these analyses; or (4) simulating a QRAM device that implements circuits in the host language itself.

Hilbert-space semantics. Although we have defined a class of normal forms for circuits, these normal forms are clearly not semantically unique. For example, consider two applications of gates g_1 and g_2 on different wires in the same circuit.

$$\mathsf{gate}\ w_1 = g_1\ w_1\ \mathsf{in}\ \mathsf{gate}\ w_2 = g_2\ w_2\ \mathsf{in}\ C$$

Morally this circuit should be equivalent to one that applies the gates in the opposite order. Such commuting conversions have been studied in quantum circuits [4, 19] and those ideas could be directly extended to a cohesive equational theory on QWIRE circuits.

Still, such a structural theory doesn't fully capture the quantum-mechanical meaning of circuits. That requires a denotational semantics of circuits as linear transformations between Hilbert spaces, in the style of Valiron [17]. Furthermore, a circuit with measurements could be interpreted as a linear transformation from its input Hilbert space to a probability distribution over its output space.

Formalization in Coq. Embedding QWIRE in the Coq programming language serves two purposes. As we demonstrated in Section 4.2, dependent types can give stronger specifications to quantum circuits like the Fourier transform. But Coq is also a proof assistant, and verifying the correctness of a quantum circuit language is a driving motivation for this line of work. Coq will allow us to directly connect the type safety and normalization theorems in this paper to an implementation of QWIRE. It will allow us to prove stronger properties, like the correctness of `reverse`, allowing us to check reversibility at compile time. Finally, we hope to prove the equivalence of programs under the semantic models discussed above and provide simple and verified translations between quantum circuits and unitary matrices.

Compilation target. QWIRE is a core circuit calculus, which means that it may not always be the best programming environment for physicists and quantum programmers looking to write code. We envision using QWIRE as a compilation target for Quipper or another quantum programming language. This would allow us to verify the type-safety of circuits while still writing programs in a rich and familiar programming environment.

References

[1] Thorsten Altenkirch & Jonathan Grattage (2005): *A functional quantum programming language*. In: *Logic in Computer Science, 2005. LICS 2005. Proceedings. 20th Annual IEEE Symposium on*, IEEE, pp. 249–258.

[2] Thorsten Altenkirch & Alexander S Green (2010): *The quantum IO monad*. *Semantic Techniques in Quantum Computation*, pp. 173–205.

[3] Stefano Bettelli, Tommaso Calarco & Luciano Serafini (2003): *Toward an Architecture for Quantum Programming*. *The European Physical Journal D* 25(2), pp. 181–200.

[4] Bob Coecke (2005): *Kindergarten quantum mechanics*. *arXiv preprint quant-ph/0510032*.

[5] Coq Development Team (2015): *The Coq Proof Assistant Reference Manual, Version 8.4*. Electronic resource, available from `http://coq.inria.fr`.

[6] Alexander Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger & Benoît Valiron (2013): *An Introduction to Quantum Programming in Quipper*. In: *Proceedings of the 5th International Conference on Reversible Computation*, *Lecture Notes in Computer Science* 7948, pp. 110–124.

[7] 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.

[8] Emmanuel H. Knill (1996): *Conventions for Quantum Pseudocode*. Technical Report LAUR-96-2724, Los Alamos National Laboratory.

[9] Neelakantan R. Krishnaswami, Pierre Pradic & Nick Benton (2015): *Integrating Linear and Dependent Types*. *SIGPLAN Notices* 50(1), pp. 17–30, doi:10.1145/2775051.2676969.

[10] Conor McBride (2016): *I got Plenty o' Nuttin'*. In: *Wadlerfest 2016*.

[11] Rodney Van Meter & Clare Horsman (2013): *A Blueprint for Building a Quantum Computer*. *Communications of the ACM* 56(10), pp. 84–93.

[12] Bernhard Ömer (2000): *Quantum Programming in QCL*. Master's thesis, Institute of Information Systems, Technical University of Vienna.

[13] Bernhard Ömer (2003): *Structured quantum programming*. *Information Systems*, p. 130.

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

[15] 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.

[16] Benoît Valiron, Neil J Ross, Peter Selinger, D Scott Alexander & Jonathan M Smith (2015): *Programming the quantum future*. *Communications of the ACM* 58(8), pp. 52–61.

[17] Benoît Valiron & Steve Zdancewic (2014): *Modeling Simply-Typed Lambda Calculi in the Category of Finite Vector Spaces*. *Scientific Annals of Computer Science* 24(2), pp. 325–368, doi:10.7561/SACS.2014.2.325.

[18] Philip Wadler (2014): *Propositions as sessions*. *Journal of Functional Programming* 24, pp. 384–418, doi:10.1017/S095679681400001X.

[19] Akira Yoshimizu, Ichiro Hasuo, Claudia Faggian & Ugo Lago (2014): *Programming Languages and Systems: 23rd European Symposium on Programming*, chapter Measurements in Proof Nets as Higher-Order Quantum Circuits, pp. 371–391. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-642-54833-8_20.

A Appendix: Collected QWIRE Semantics

$$
\begin{array}{ll}
\textit{(wire patterns)} & p ::= w \mid \mathsf{Q}\,w \mid \mathsf{B}\,w \mid () \mid (p_1, p_2) \\
\textit{(wire types)} & W ::= \mathsf{qubit} \mid \mathsf{bit} \mid 1 \mid W_1 \otimes W_2 \\
\textit{(unitaries)} & u ::= \mathsf{H} \mid \mathsf{QNOT} \mid \mathsf{C}\,u \mid \mathsf{CC}\,u \mid u^\dagger \\
\textit{(gates)} & g ::= u \mid \mathsf{qinit}_0 \mid \mathsf{qinit}_1 \mid \mathsf{cinit}_0 \mid \mathsf{cinit}_1 \mid \mathsf{meas} \mid \mathsf{cdiscard} \mid \mathsf{cdup} \\
\textit{(circuits)} & C ::= p_1 \leftrightarrow p_2 \mid \mathsf{gate}\, p_2 = g\, p_1 \,\mathsf{in}\, C \mid \mathsf{unbox}\, p_2 = t\, p_1 \,\mathsf{in}\, C \mid \langle C_1 \mapsto C_2 \rangle \\
\textit{(terms)} & t ::= \cdots \mid \mathsf{box}\,(p_1 \Rightarrow C)\,\mathsf{return}\,p_2 \\
\textit{(classical types)} & A ::= \cdots \mid \mathsf{CIRC}(W_1, W_2)
\end{array}
$$

$\boxed{\Gamma \vdash t : A}$

$$\frac{\Omega_1 \vdash p_1 : W_1 \quad \Gamma; \Omega_1 \vdash C \mid \Omega_2 \quad \Omega_2 \vdash p_2 : W_2}{\Gamma \vdash \mathsf{box}\,(p_1 \Rightarrow C)\,\mathsf{return}\,p_2 : \mathsf{CIRC}(W_1, W_2)}$$

$\boxed{\Gamma; \Omega_1 \vdash C \mid \Omega_2}$

$$\frac{\Omega_1 \vdash p_1 : \Sigma_{\mathsf{In}}(g) \quad \Omega_2 \vdash p_2 : \Sigma_{\mathsf{Out}}(g) \quad \Gamma; \Omega_0, \Omega_2 \vdash C \mid \Omega}{\Gamma; \Omega_0, \Omega_1 \vdash \mathsf{gate}\, p_2 = g\, p_1 \,\mathsf{in}\, C \mid \Omega}$$

$$\frac{\Omega_1 \vdash p_1 : W \quad \Omega_2 \vdash p_2 : W}{\Gamma; \Omega_1 \vdash p_1 \leftrightarrow p_2 \mid \Omega_2} \qquad \frac{\Gamma; \Omega_1 \vdash C_1 \mid \Omega_1', \Omega \quad \Gamma; \Omega_2, \Omega \vdash C_2 \mid \Omega_2'}{\Gamma; \Omega_1, \Omega_2 \vdash \langle C_1 \mapsto C_2 \rangle \mid \Omega_1', \Omega_2'}$$

$$\frac{\Omega_1 \vdash p_1 : W_1 \quad \Gamma \vdash t : \mathsf{CIRC}(W_1, W_2) \quad \Omega_2 \vdash p_2 : W_2 \quad \Gamma; \Omega_0, \Omega_2 \vdash C \mid \Omega}{\Gamma; \Omega_0, \Omega_1 \vdash \mathsf{unbox}\, p_2 = t\, p_1 \,\mathsf{in}\, C \mid \Omega}$$

$\boxed{\Omega \vdash p : W}$

$$\frac{}{w : W \vdash w : W} \qquad \frac{}{w : \mathsf{qubit} \vdash \mathsf{Q}\,w : \mathsf{qubit}} \qquad \frac{}{w : \mathsf{bit} \vdash \mathsf{B}\,w : \mathsf{bit}}$$

$$\frac{}{\cdot \vdash () : 1} \qquad \frac{\Omega_1 \vdash p_1 : W_1 \quad \Omega_2 \vdash p_2 : W_2}{\Omega_1, \Omega_2 \vdash (p_1, p_2) : W_1 \otimes W_2}$$

$\boxed{(W_1, g, W_2) \in \Sigma}$

u	$\Sigma_{\mathsf{In}}(u)$	$\Sigma_{\mathsf{Out}}(u)$
H	qubit	qubit
QNOT	qubit	qubit
$u^\dagger$	$\Sigma_{\mathsf{Out}}(u)$	$\Sigma_{\mathsf{In}}(u)$
C u	qubit $\otimes\, \Sigma_{\mathsf{In}}(u)$	qubit $\otimes\, \Sigma_{\mathsf{Out}}(u)$
CC u	bit $\otimes\, \Sigma_{\mathsf{In}}(u)$	bit $\otimes\, \Sigma_{\mathsf{Out}}(u)$

g	$\Sigma_{\mathsf{In}}(g)$	$\Sigma_{\mathsf{Out}}(g)$
$\mathsf{qinit}_0, \mathsf{qinit}_1$	1	qubit
$\mathsf{cinit}_0, \mathsf{cinit}_1$	1	bit
meas	qubit	bit
cdiscard	bit	1
cdup	bit	bit $\otimes$ bit

Figure 4: The QWIRE circuit language static semantics.

$\boxed{t \Longrightarrow t'}$

$$\frac{C \Longrightarrow C'}{\mathsf{box}\ (n_1 \Rightarrow C)\ \mathsf{return}\ n_2 \Longrightarrow \mathsf{box}\ (n_1 \Rightarrow C')\ \mathsf{return}\ n_2}$$

$\boxed{t : A \Longrightarrow_\eta t'}$

$$\frac{\mathcal{Q}_1 \vdash n_1 : W_1 \quad \mathcal{Q}_2 \vdash n_2 : W_2}{t : \mathsf{CIRC}(W_1, W_2) \Longrightarrow_\eta \mathsf{box}\ (n_1 \Rightarrow \mathsf{unbox}\ w = t\ n_1\ \mathsf{in}\ w \leftrightarrow n_2)\ \mathsf{return}\ n_2}$$

$\boxed{C_1 : W \Longrightarrow C_2}$

$$\frac{\mathcal{Q}_1 \vdash n_1 : W}{p_1 \leftrightarrow p_2 : W \Longrightarrow n_1 \leftrightarrow p_2}$$

$\boxed{C_1 \Longrightarrow C_2}$

$$\frac{C \Longrightarrow C'}{\mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ C \Longrightarrow \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ C'}$$

$$\frac{\mathcal{Q}_1 \vdash n_1 : \Sigma_{\mathsf{In}}(g) \quad \mathcal{Q}_2 \vdash n_2 : \Sigma_{\mathsf{Out}}(g)}{\mathsf{gate}\ p_2 = g\ p_1\ \mathsf{in}\ C \Longrightarrow \mathsf{gate}\ n_2 = g\ p_1\ \mathsf{in}\ \langle n_2 \leftrightarrow p_2 \mapsto C\rangle}$$

$$\frac{t \Longrightarrow t'}{\mathsf{unbox}\ p_2 = t\ p_1\ \mathsf{in}\ C \Longrightarrow \mathsf{unbox}\ p_2 = t'\ p_1\ \mathsf{in}\ C}$$

$$\overline{\mathsf{unbox}\ p_2 = (\mathsf{box}\ (p_1' \Rightarrow C_1)\ \mathsf{return}\ p_2')\ p_1\ \mathsf{in}\ C_2 \Longrightarrow \langle\langle p_1 \leftrightarrow p_1' \mapsto \langle C_1 \mapsto p_2' \leftrightarrow p_2\rangle\rangle \mapsto C_2\rangle}$$

$$\frac{C_1 \Longrightarrow C_1'}{\langle C_1 \mapsto C_2\rangle \Longrightarrow \langle C_1' \mapsto C_2\rangle}$$

$$\overline{\langle \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ N_1 \mapsto C_2\rangle \Longrightarrow \mathsf{gate}\ n_2 = g\ n_1\ \mathsf{in}\ \langle N_1 \mapsto C_2\rangle}$$

$$\frac{\mathrm{output}(n_1 \leftrightarrow p_2) \perp \mathrm{input}(C_2) \quad C_2 \Longrightarrow C_2'}{\langle n_1 \leftrightarrow p_2 \mapsto C_2\rangle \Longrightarrow \langle n_1 \leftrightarrow p_2 \mapsto C_2'\rangle}$$

$$\frac{\mathrm{output}(n_1 \leftrightarrow p_2) \perp \mathrm{input}(\mathsf{gate}\ n_2' = g\ n_1'\ \mathsf{in}\ C_2)}{\langle n_1 \leftrightarrow p_2 \mapsto \mathsf{gate}\ n_2' = g\ n_1'\ \mathsf{in}\ C_2\rangle \Longrightarrow \mathsf{gate}\ n_2' = g\ n_1'\ \mathsf{in}\ \langle n_1 \leftrightarrow p_2 \mapsto C_2\rangle}$$

$$\frac{\mathrm{output}((n_1', n_2') \leftrightarrow (p_1', p_2')) \not\perp \mathrm{input}(C)}{\langle (n_1', n_2') \leftrightarrow (p_1', p_2') \mapsto C\rangle \Longrightarrow \langle n_1' \leftrightarrow p_1' \mapsto \langle n_2' \leftrightarrow p_2' \mapsto C\rangle\rangle}$$

$$\frac{w_2 \in \mathrm{input}(C)}{\langle \mathsf{Q}\, w_1 \leftrightarrow \mathsf{Q}\, w_2 \mapsto C\rangle \Longrightarrow C\{w_1/w_2\}} \qquad \frac{w_2 \in \mathrm{input}(C)}{\langle \mathsf{B}\, w_1 \leftrightarrow \mathsf{B}\, w_2 \mapsto C\rangle \Longrightarrow C\{w_1/w_2\}}$$

$$\frac{w_2 \in \mathrm{input}(C)}{\langle n_1 \leftrightarrow w_2 \mapsto C\rangle \Longrightarrow C\{n_1/w_2\}} \qquad \frac{\mathrm{output}(n_1 \leftrightarrow p_2) \perp \mathrm{input}(n_1' \leftrightarrow p_2')}{\langle n_1 \leftrightarrow p_2 \mapsto n_1' \leftrightarrow p_2'\rangle \Longrightarrow (n_1, n_1') \leftrightarrow (p_2, p_2')}$$

Figure 5: The $\mathcal{Q}$WIRE circuit language dynamic semantics.