QWIRE

A Core Language for Quantum Circuits
Contents

- **Context**: interaction between classical and quantum computers
- Qwire introduction with **examples**
- **Type System** for well-formed circuits
- **Operational Semantics** for circuit normalization
- **Denotational Semantics** based on density matrices
- **Extensions and Applications** (Quantum Oracles)
QRAM model of quantum computing

Host Language

- Circuits as host types
- Guarantee that circuits are well-formed
- Still allowing abstractions and high level features
A Minimal Core

Host Language is **parametric**

- Could be **instantiated** with a wide range of programming languages
  - High-level functional programming languages
  - Theorem provers (e.g. Coq)

- Only the **interaction** between classical and quantum computer is formalized
Guarantee Circuits Safety

Using a strong type system

- We need **linear types** (qubits cannot be duplicated)
- But in the host language we also need **non-linear types**
- Integrating linear types in existing languages is (very) difficult
  - Linear types for the circuit language
  - Non-linear types for the host language
- Runtime errors arise only from host language!
Box and Unboxing

Boxed circuits
- In the host language
- Non-linear types
- Can be used inside the host language

Unboxed circuits
- In the circuit language
- Linear types
- Can be reused inside other circuits

Box and Unbox rules link the type systems of Host and Qwire
Some Examples
A First Example

\[
\text{hadamard-measure} : \text{Circ}(\text{qubit}, \text{bit}) = \\
\text{box } w \Rightarrow \\
w' \gets \text{gate } H \ w; \\
b \gets \text{gate } \text{meas } w'; \\
\text{output } b
\]

\text{Circ}(W, W) \text{ is the (Host) type of circuits}

- $\bar{w}$ is a wire type (bit/qubit wire and their composition)
- $w$ is a wire name, it is not a regular variable (it is linear)
A Wrong Example

\[
\text{absurd} = \text{box } w \Rightarrow \\
x \leftarrow \text{gate meas } w; \\
w' \leftarrow \text{gate H } w; \\
\text{output } (x,w')
\]

\begin{itemize}
  \item $w$ cannot be used two times
  \item Similar property: qubits cannot be discarded implicitly
    \begin{itemize}
      \item You have to explicitly discard them (after a measurement)
    \end{itemize}
\end{itemize}
Composing Gates

Gates act on wires, not on circuits

- gate meas (gate H w) is ill-formed
- Gates can be composed by connecting wires
  e.g.
  
  ```
  w' <- gate H w;
  b  <- gate meas w';
  ```

Same for circuits

- But they must be unboxed (connecting a wire to the input)
  e.g.
  
  ```
  w2  <- unbox c1 w1;
  unbox c2 w2
  ```
Sequential and Parallel Composition of Circuits

\[
\text{inSeq} \ (c_1 : \text{Circ}(W_1,W_2)) \ (c_2 : \text{Circ}(W_2,W_3))
\]
\[
: \text{Circ}(W_1,W_3) = \text{box} \ w_1 \Rightarrow
\]
\[
w_2 \leftarrow \text{unbox} \ c_1 \ w_1;
\]
\[
\text{unbox} \ c_2 \ w_2
\]

\[
\text{inPar} \ (c : \text{Circ}(W_1,W_2)) \ (c' : \text{Circ}(W_1',W_2'))
\]
\[
: \text{Circ}(W_1 \otimes W_1', \ W_2 \otimes W_2') =
\]
\[
\text{box} \ (w_1,w_1') \Rightarrow
\]
\[
w_2 \leftarrow \text{unbox} \ c \ w_1;
\]
\[
w_2' \leftarrow \text{unbox} \ c' \ w_1';
\]
\[
\text{output} \ (w_2,w_2')
\]

In both cases the type system guarantees that the wire types match
Dynamic Lifting - example with Quantum Teleportation
Bob with Dynamic Lifting

\[
\text{bob} : \text{Circ}(\text{bit} \otimes \text{bit} \otimes \text{qubit}, \text{qubit}) = \\
\text{box} (x, y, b) \Rightarrow \\
(y, b) \leftarrow \text{gate (bit-control X)} (y, b) ; \\
(x, b) \leftarrow \text{gate (bit-control Z)} (x, b) ; \\
() \leftarrow \text{gate discard y} ; \\
() \leftarrow \text{gate discard x} ; \\
\text{output} b
\]
Bob with Dynamic Lifting

bob : Circ(bit⊗bit⊗qubit, qubit) =
  box (x,y,b) =>
    (y,b) <- gate (bit-control X) (y,b);
    (x,b) <- gate (bit-control Z) (x,b);
    () <- gate discard y;
    () <- gate discard x;
  output b

bob-dyn : Circ(bit⊗bit⊗qubit, qubit) =
  box (w1,w2,q) =>
    (x1,x2) <- lift (w1,w2);
    q <- unbox (if x2 then X_gate else id) q;
    unbox (if x1 then Z_gate else id) q
Running a Circuit

flip : Bool =
    run (q <- gate init0 ();
         q <- gate H q;
         b <- gate meas q;
         output b)

run operation

- Take a circuit with no input (wire of type 1, with only value () )
- Returns a host value
Qwire Type System
Basic Ingredients

Wire types \( W ::= 1 \mid \text{bit} \mid \text{qubit} \mid W_1 \otimes W_2 \)

Gates have input and output wire type, and we assume

- If a unitary \( u \in \mathcal{G}(W, W) \) then
  \[
  u^\dagger \in \mathcal{G}(W, W)
  \]
  \[
  \text{control } u \in \mathcal{G}(\text{qubit} \otimes W, \text{qubit} \otimes W)
  \]
  \[
  \text{bit-control } u \in \mathcal{G}(\text{bit} \otimes W, \text{bit} \otimes W)
  \]

- Initialization \( \text{new0, new1} \in \mathcal{G}(1, \text{bit}), \text{init0, init1} \in \mathcal{G}(1, \text{qubit}) \)
- Measurement and discard \( \text{meas} \in \mathcal{G}(\text{qubit, bit}), \text{discard} \in \mathcal{G}(\text{bit, 1}) \)
Typing Judgements for well-formed circuits

Judgement $\Gamma; \Omega \vdash C : W$ with

- $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ context of \textbf{host} variables with host types
- $\Omega = w_1 : W_1, \ldots, w_n : W_n$ context of \textbf{wire} variables with wire types
- $C$ a circuit
- $W$ the output wire type
Auxiliary Judgement

Judgement $\Omega \Rightarrow p : W$ defined as

\[
\cdot \Rightarrow () : 1 \\
\]

$w : W \Rightarrow w : W$

\[
\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \\
\Omega_1, \Omega_2 \Rightarrow (p_1, p_2) : W_1 \otimes W_2
\]
Auxiliary Judgement

Note that $\Omega \Rightarrow p : W$ is linear

E.g. the following judgements do not hold

- $w : W, w' : W' \Rightarrow w : W$
- $w : W \Rightarrow (w, w) : W \otimes W$
Assumptions on the Host Language

We will assume types $A$ are such that there is at least:

- a corresponding type that is the lifting of each wire type
  - bit and qubits (booleans)
  - tensor product (pairs)

\[
|\text{bit}| = \text{Bool} \quad |1| = \text{Unit} \\
|\text{qubit}| = \text{Bool} \quad |W_1 \otimes W_2| = |W_1| \times |W_2|
\]

- a type for circuits with typed input and output $\text{Circ}(W_1, W_2)$
Typing Rules for Qwire

Output

- Build a pattern of its input wires

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

\[\text{OUTPUT}\]

\[\Omega \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad W\]
Typing Rules for Qwire

Gates

- Are applied to a pattern of wires
- The output defines another pattern
- Unused wires and output wires are used in the continuation

\[
g \in \mathcal{G}(W_1, W_2) \\
\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \Gamma; \Omega_2, \Omega \vdash C : W \\
\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \ p_1 ; C : W
\]
Typing Rules for Qwire

Composition

- Same as gates application
- ...
- But the correctness of the input types is defined recursively

\[
\frac{\Gamma; \Omega_1 \vdash C : W \quad \Omega \Rightarrow p : W \quad \Gamma; \Omega, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'}
\]

COMPOSE
Typing Rules for Qwire

Boxing

- Bridge from Qwire circuits to Host terms
  - Qwire type above
  - Host type below

\[
\Omega \Rightarrow p : W_1 \quad \Gamma ; \Omega \vdash C : W_2
\]

\[
\Gamma \vdash \text{box} (p : W_1) \Rightarrow C : \text{Circ}(W_1, W_2)
\]

BOX
Typing Rules for Qwire

Unboxing

- Bridge from Host terms to Qwire circuits
  - Host type above
  - Qwire type below

\[
\Gamma \vdash t : \text{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1 \\
\Gamma; \Omega \vdash \text{unbox} \ t \ p : W_2 \\
\text{UNBOX}
\]
Typing Rules for Qwire

Running a circuit

- Host can run Qwire circuits
  - When the circuit has no input wires
  - Qwire type above
  - Host type below

\[ |\text{bit}| = \text{Bool} \]
\[ |\text{qubit}| = \text{Bool} \]
\[ |1| = \text{Unit} \]
\[ |W_1 \otimes W_2| = |W_1| \times |W_2| \]

\[ \frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \text{run} C : |W|} \quad \text{RUN} \]
Typing Rules for Qwire

Lifting a wire

- Qwire can measure a wire and use the result in Host
  - We update the host context
  - And we continue with the judgement

\[
\Omega \Rightarrow p : W \quad \Gamma, x : |W|; \Omega' \vdash C : W' \\
\overline{\Gamma; \Omega, \Omega' \vdash x \leftarrow \text{lift } p; C : W'} \\
\text{LIFT}
\]
Static VS Dynamic Lifting

- Run is a **static lifting** operator
  - All the wires are measured (or discarded)
  - No residual state is left on the quantum computer

- Lift is a **dynamic lifting** operator
  - Only a subset of the wires are measured
  - The classical computer uses the result to compute the rest of the circuit
  - The state of the quantum computer must be preserved
Circuit Normalization
Operational Semantics

- Circuits represents instructions for the quantum computer
- Composition and unbox are meta-operations
- The (small-step) operational semantics normalizes circuits

\[ C \implies C' \]

- Eliminates unboxing and composition
- Concretizes patterns (no tuples of wires)

\[ \cdot; Q \vdash C : W \quad \text{where} \quad Q ::= \cdot \mid Q, w : \text{bit} \mid Q, w : \text{qubit} \]

\[ N ::= \text{output } p \mid p_2 \leftarrow \text{gate } g \ p_1; N \mid x \leftarrow \text{lift } p; C \]
Operational Semantics - ingredients

- Pattern generalization \( p' \preceq p \) (e.g. \( p \preceq w \))
- Concrete patterns \( p \) for \( W \), i.e. s.t. for all \( \Omega \Rightarrow p' : W \) it is \( \neg(p' \prec p) \)
  - example of concretization:
    - from \( w : \text{bit} \otimes \text{bit} \Rightarrow w : \text{bit} \otimes \text{bit} \)
    - to \( w' : \text{bit}, w : \text{bit} \Rightarrow (w', w) : \text{bit} \otimes \text{bit} \)
- Host terms evaluation \( t \rightarrow t' \) is the union of
  - Host language alone \( \rightarrow_H \)
  - Boxed circuits \( \rightarrow_b \)
Operational Semantics

Box

- Concretizes the pattern first
- Then normalizes the circuit

\[
\begin{align*}
p \text{ is concrete for } W & \quad C \quad \Rightarrow \quad C' \\
\text{box} (p : W) \Rightarrow C & \quad \overset{b}{\longrightarrow} \quad \text{box} (p : W) \Rightarrow C'
\end{align*}
\]

\[
\begin{align*}
p' & \prec p \\
p' \text{ is concrete for } W \\
\text{box} (p : W) \Rightarrow C & \quad \overset{b}{\longrightarrow} \quad \text{box} p' \Rightarrow C \{p'/p\}
\end{align*}
\]

\[\eta\]
Operational Semantics

Unbox

- Just reduces to the terms evaluation
- And eliminates unbox-box pairs

\[
\frac{t \rightarrow t'}{\text{unbox } t \ p \ \Rightarrow \ \text{unbox } t' \ p} \quad \text{STRUCT}
\]

\[
\frac{\text{unbox } (\text{box } (p : W) \Rightarrow N) \ p' \ \Rightarrow \ N \ \{p' / p\} \quad \beta}{\beta}
\]
Operational Semantics

Gate

- Concretizes the pattern first
- Then proceeds with the continuation

\[
g \in G(W_1, W_2) \quad p_2 \text{ is concrete for } W_2 \quad C \implies C' \\
p_2 \leftarrow \text{gate } g \ p_1; \ C \implies p_2 \leftarrow \text{gate } g \ p_1; \ C'
\]

\[
g \in G(W_1, W_2) \quad p_2' \prec p_2 \quad p_2' \text{ is concrete for } W_2 \\
p_2 \leftarrow \text{gate } g \ p_1; \ C \implies p_2' \leftarrow \text{gate } g \ p_1; \ C \ \{p_2'/p_2\}
\]
Operational Semantics

Composition

- Normalizes the circuits in order
- Substitutes patterns when associated with outputs
- ...

\[ C_1 \implies C'_1 \]
\[ p \leftarrow C_1; C_2 \implies p \leftarrow C'_1; C_2 \quad \text{STRUCT} \]

\[ p \leftarrow \text{output } p'; C \implies C \{ p'/p \} \quad \beta \]
Operational Semantics

Composition

- ...
- Postpones the connection of wires after gate and lifting operations (Commuting Conversion)

\[
p \leftarrow (p_2 \leftarrow \text{gate } g \ p_1; \ N); \ C \Rightarrow p_2 \leftarrow \text{gate } g \ p_1; p \leftarrow N; \ C \quad \text{CC}
\]

\[
p' \leftarrow (x \leftarrow \text{lift } p; \ C'); \ C \Rightarrow x \leftarrow \text{lift } p; p' \leftarrow C'; \ C' \quad \text{CC}
\]
Operational Semantics Properties

The normalization satisfies

- **Preservation**
  - Same type before and after reduction

- **Progress**
  - If not in normal form then a next step exists

- **Normalization**
  - Normal form is always reachable

... assuming that also $\rightarrow_H$ satisfies them
Denotational Semantics
Why a Denotational Semantics?

Because we want to

- Specify the actual **physical meaning** of the language
  - Nothing unexpected

- Prove **soundness** of the operational semantics
  - The denotational semantics of a circuit is the same of its normalization
Denotational Semantics

- We have to deal with **ordering** and permutations
  - Qwire contexts are unordered
  - Elements of an Hilbert space are ordered
  - We will consider ordered contexts with explicit permutations
  - Then we can simply use

\[
\begin{align*}
[\cdot] &= \mathcal{H}_1 \\
[w : W] &= [W] \\
[\Omega_1, \Omega_2] &= [\Omega_1] \otimes [\Omega_2]
\end{align*}
\]

with:

\[
\begin{align*}
[\text{bit}] &= \mathcal{H}_2 \\
[1] &= \mathcal{H}_1 \\
[qubit] &= \mathcal{H}_2 \\
[W_1 \otimes W_2] &= [W_1] \otimes [W_2]
\end{align*}
\]
Denotational Semantics

- Semantics of values is as expected

for $[v : \mid W \mid]$ we have an elemento of $[W]$

$$[[\ast : \text{Unit}]] = \mid \ast \rangle$$
$$[[\text{false} : \text{Bool}]] = \mid 0 \rangle$$
$$[[\text{true} : \text{Bool}]] = \mid 1 \rangle$$

$$[[\langle v_1, v_2 \rangle : \mid W_1 \mid \times \mid W_2 \mid]] = [[v_1 : \mid W_1 \mid]] \otimes [[v_2 : \mid W_2 \mid]]$$
Denotational Semantics

- Gates and circuits are represented as super-operators

For $g \in \mathcal{G}(W_1, W_2)$ we have $[[g]]$ is a super operator from $W_1$ to $W_2$

$$[[\text{new0}], [\text{init0}]] = (|0\rangle \langle 0|)^*$$

$$[[\text{new1}], [\text{init1}]] = (|1\rangle \langle 1|)^*$$

$$[[\text{meas}]] = (|0\rangle \langle 0|)^* + (|1\rangle \langle 1|)^*$$

$$[[\text{discard}]] = \langle 0|^* + \langle 1|^*$$

where $f^* \rho = f \rho f^\dagger$
Denotational Semantics

\[
\begin{align*}
\Omega \Rightarrow p : W \\
\therefore \Omega \vdash \text{output } p : W \\
\vdots \Omega' \vdash C : W \quad \pi : \Omega \equiv \Omega' \\
\therefore \Omega \vdash C : W \\
\vdots \vdash t : \text{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1 \\
\therefore \Omega \vdash \text{unbox } t \; p : W_2 \\
g \in \mathcal{G}(W_1, W_2) \\
\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \vdots \Omega_2, \Omega \vdash C : W \\
\therefore \Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \; p_1 ; C : W \\
\vdots \Omega \Rightarrow p : W \quad x : |W| ; \Omega' \vdash C : W' \\
\therefore \Omega, \Omega' \vdash x \leftarrow \text{lift } p ; C : W' \\
\vdots \vdash \Omega_1 \vdash C : W' \quad \Omega_0 \Rightarrow p : W \quad \vdots \Omega_0, \Omega_2 \vdash C' : W' \\
\therefore \Omega_1, \Omega_2 \vdash p \leftarrow C ; C' : W' \end{align*}
\]

\[[\Omega \vdash \text{output } p : W] = I^*\]

\[[\Omega \vdash C : W] = [[\Omega' \vdash C : W] \circ [\pi]^*\]

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

\[[\Omega_1, \Omega \vdash p_2 \leftarrow \text{gate } g \; p_1 ; C : W] = [[\Omega_2, \Omega \vdash C : W] \circ ([g] \otimes I^*)]\]

\[[\Omega, \Omega' \vdash x \leftarrow \text{lift } p ; C : W'] = \sum_{\vdash v : |W|} [[\Omega' \vdash C\{v/x\} : W'] \circ ([v : |W|] \uparrow \otimes I^*)]\]

\[[\Omega_1, \Omega_2 \vdash p \leftarrow C ; C' : W'] = [[\Omega_0, \Omega_2 \vdash C' : W'] \circ ([\Omega_1 \vdash C : W] \otimes I^*)\]
Using Qwires
Extensions

- Qwire is a minimal **core** language
- Its strength is that it allows **extensions**
- We will see a pair of them
  - Pattern Matching on Circuits
  - Dependent Types
  - ReQwire for reasoning about reversible circuits
Pattern Matching

- We can write a host-level representation of patterns and gates
- Inductive data structure equivalent to $\text{Circ}(W_1, W_2)$
  
  \[
  \begin{array}{l}
  \text{type ICirc } W_1 W_2 = \\
  | \text{Output} : \text{Pat } W_1 W_2 \rightarrow \text{ICirc } W_1 W_2 \\
  | \text{Gate} : \text{Pat } W_1 (W_1' \otimes W_0) \rightarrow \text{Gate } W_1' W_2' \rightarrow \\
  \hspace{1cm} \text{Circ}(W_2' \otimes W_0, W_2) \rightarrow \text{ICirc } W_1 W_2 \\
  | \text{Lift} : \text{Pat } W_1 (W \otimes W') \rightarrow \\
  \hspace{1cm} (|W| \rightarrow \text{Circ}(W',W_2)) \rightarrow \text{ICirc } W_1 W_2.
  \end{array}
  \]

- Functions from ICirc to Circ and vice-versa
- With this we can do pattern matching on circuits inside Host!
Pattern Matching

- We can write a function that safely revert circuits

```haskell
reverse (c : Circ(W1,W2)) : Option (Circ(W2,W1)) =
  case toICirc c of
  | Output p -> fromICirc (Output (reverse_pat p))
  | Gate p g c' ->
    case reverse (toICirc c'), reverse_gate g of
    | Some c_rev, Some g_rev ->
      let p_rev = reverse_pat p in
      let i_rev = Gate id_pat g_rev (Output p_rev) in
      inSeq c_rev (fromICirc i_rev)
    | _, _ -> None
  end
  | Lift _ _ -> None
end
```
Dependent Types

Note that the number of wires of a circuit is part of its type

- We would like to have functions that generate circuits with a number of wires that depends on the input, i.e. dependent types

Combining dependent and linear types is active research

- But Qwire keeps linear and non-linear types separated
- Types will depend only on non-linear terms
Dependent Types

E.g., the following function returns a circuit with wire types that depends on the inputs

```
rotations (m:Nat) : \Pi (n:Nat).
  CIRC(\otimes (n+1) qubit, \otimes (n+1) qubit) =
  fun n => case n of
    | 0   -> id
    | 1   -> id
    | S n' -> box (c,(q,qs)) =>
      (c,qs) <- unbox rotations m n' (c,qs);
      (c,q) <- gate (control (RGate (2+m-n'))) (c,q);
      output (c,(q,w))
  end
```
ReQwire: Reasoning about Reversible Circuits

Quantum algorithms commonly use **quantum oracles**

- Reversible logic circuits require ancillae
- After usage, we want to discard them
- We must be sure of their state to do that
  - Measuring an entangled qubit affects the result!

**ReQwire** allows to recognise syntactically valid ancillae

- Allowing the definition of a compiler for oracles
ReQwire: Reasoning about Reversible Circuits

E.g., the following circuit does not make a correct use of ancillae

\[z \oplus ((a \lor b) \land (c \lor d))\]
We add assertion gates that discard a qubit in the given state:

\[ g := U \mid \text{init}_0 \mid \text{init}_1 \mid \text{meas} \mid \text{discard} \mid \text{assert}_0 \mid \text{assert}_1 \]

We give a pair of denotational semantics:

- **Safe semantics** (measures the qubit before discarding)
- **Unsafe semantics** (trusts the assertion and just discards)
  - producing an illegal matrix if the assertion is wrong

A circuit is **valid** (all the assertions are correct) if the two **agree**
Based on the denotational precise definition of validity

- A syntactic property called **source symmetry** is defined for circuits with classical gates
- It is proved to be a **sufficient condition** for validity
- Source symmetric circuits are characterized inductively
- (A compiler for source symmetric (thus valid) oracles is given)
ReQwire: Reasoning about Reversible Circuits

Classical Gates:

- Initialization gates
- Assertion gates
- Not gate
- Controlled not gate
- Toffoli gate
ReQwire: Reasoning about Reversible Circuits

Definition of **Source symmetric circuits**

The input wires of a circuit are divided in

- **N source** qubits - input of the boolean function
- **1 target** qubit     - output of the boolean function

Source symmetric circuits behaves as the **identity on source qubits**

**Idea for the characterization:** they must **uncompute** the value obtained for the source qubits (using the inverse)

(Note that the inverse of a classical gate is itself)
ReQwire: Reasoning about Reversible Circuits

Roughly

- The identity is source symmetric
- If \( g \) classical and \( c \) source symmetric then
  - \( g ;; c ;; g \) is source symmetric
- If \( g \) classical only acts on target and \( c \) is source symmetric
  - \( g ;; c \) and \( c ;; g \) are source symmetric
- If \( c \) is source symmetric and \( i \) is in its source then
  - \( (\text{init}_b \text{ at } i) ;; c ;; (\text{assert}_b \text{ at } i) \) is source symmetric
ReQwire: Reasoning about Reversible Circuits

Compiling Oracles

- Given a boolean expression
  \[ b ::= x \mid t \mid f \mid \neg b \mid b_1 \land b_2 \mid b_1 \oplus b_2 \]
- and a map \( \Gamma \) from variables to wire indices
- Returns a circuit with
  - a wire for each source variable
  - a target wire for the result
ReQwire: Reasoning about Reversible Circuits

Compiling Oracles

$\text{compile } b \Gamma$

Target

Source wires

$[b] \oplus \text{Target}$

Unchanged source wires
ReQwire: Reasoning about Reversible Circuits

Compiling a variable, true and false

\[ b ::= x | t | f | \neg b | b_1 \land b_2 | b_1 \oplus b_2 \]
ReQwire: Reasoning about Reversible Circuits

Compiling a negation

\[ b ::= x \mid t \mid f \mid \neg b \mid b_1 \land b_2 \mid b_1 \oplus b_2 \]
ReQwire: Reasoning about Reversible Circuits

Compiling a conjunction

\[ b ::= x \mid t \mid f \mid \neg b \mid b_1 \land b_2 \mid b_1 \oplus b_2 \]
ReQwire: Reasoning about Reversible Circuits

Compiling an exclusive disjunction

\[ b ::= x \mid t \mid f \mid \neg b \mid b_1 \land b_2 \mid b_1 \oplus b_2 \]

here we have 0
ReQwire: Reasoning about Reversible Circuits

Correctness of the compilation (in Coq)

Theorem compile_correct :

\forall (b : bexp) (\Gamma : Ctx) (f : Var \to bool) (z : bool),
vars b \subseteq \text{domain } \Gamma \to
[\text{compile } b \ \Gamma] (\text{bool_to_matrix } z \otimes \text{basis_state } \Gamma f) =
\text{bool_to_matrix } (z \oplus [b]_f) \otimes \text{basis_state } \Gamma f.
Conclusion

Qwire gives a simple, parametric description of the minimal core for a system in which classical and quantum computations interact.

Has good properties

- Normalization
- Static typing guarantee runtime errors only due to Host
- Formal denotational semantics

Allows for interesting extensions and applications

- Pattern matching, dependent types, valid oracles
Bibliography
