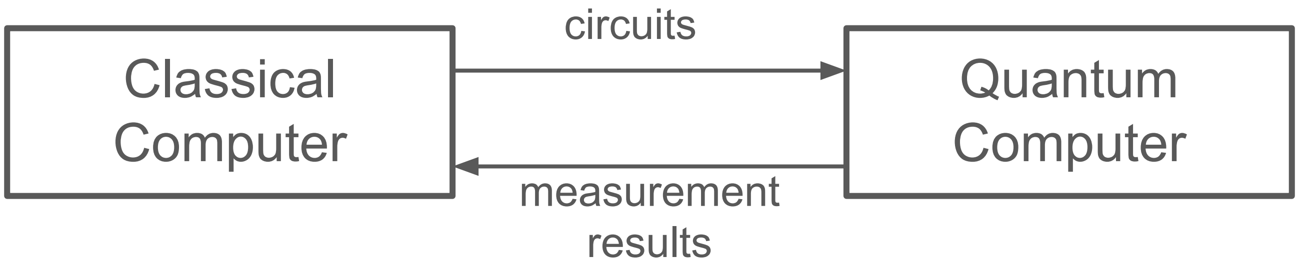
QWIRE

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

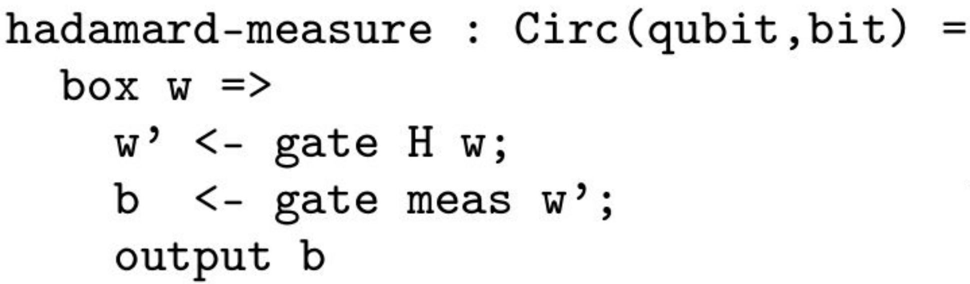
Unboxed circuits

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

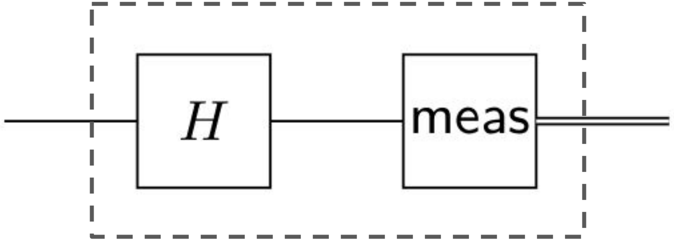
Box and Unbox rules link the type systems of Host and Qwire

Some Examples

A First Example



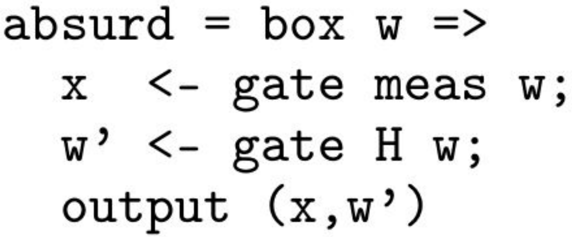
```
hadamard-measure : Circ(qubit,bit) =
  box w =>
    w’ <- gate H w;
    b  <- gate meas w’;
    output b
```



`Circ(W, W)` is the (Host) type of circuits

- `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



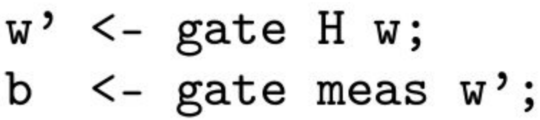
```
absurd = box w =>
  x  <- gate meas w;
  w’ <- gate H w;
  output (x,w’)
```

- `w` cannot be used two times
- Similar property: qubits cannot be discarded implicitly
  - You have to explicitly discard them (after a measurement)

Composing Gates

Gates act on wires, not on circuits

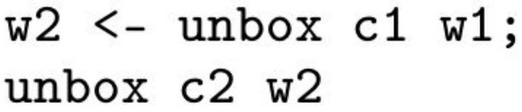
- `gate meas (gate H w)` is ill-formed
- Gates can be composed by connecting wires
  e.g.



```
w' <- gate H w;
b  <- gate meas w';
```

Same for circuits

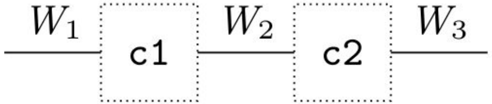
- But they must be unboxed (connecting a wire to the input)
  e.g.



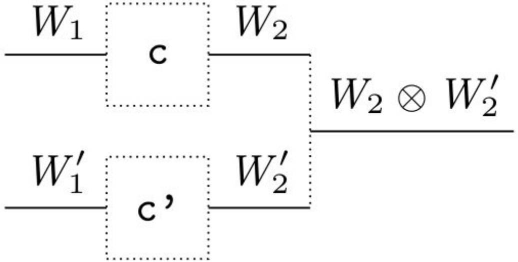
```
w2 <- unbox c1 w1;
unbox c2 w2
```

Sequential and Parallel Composition of Circuits

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

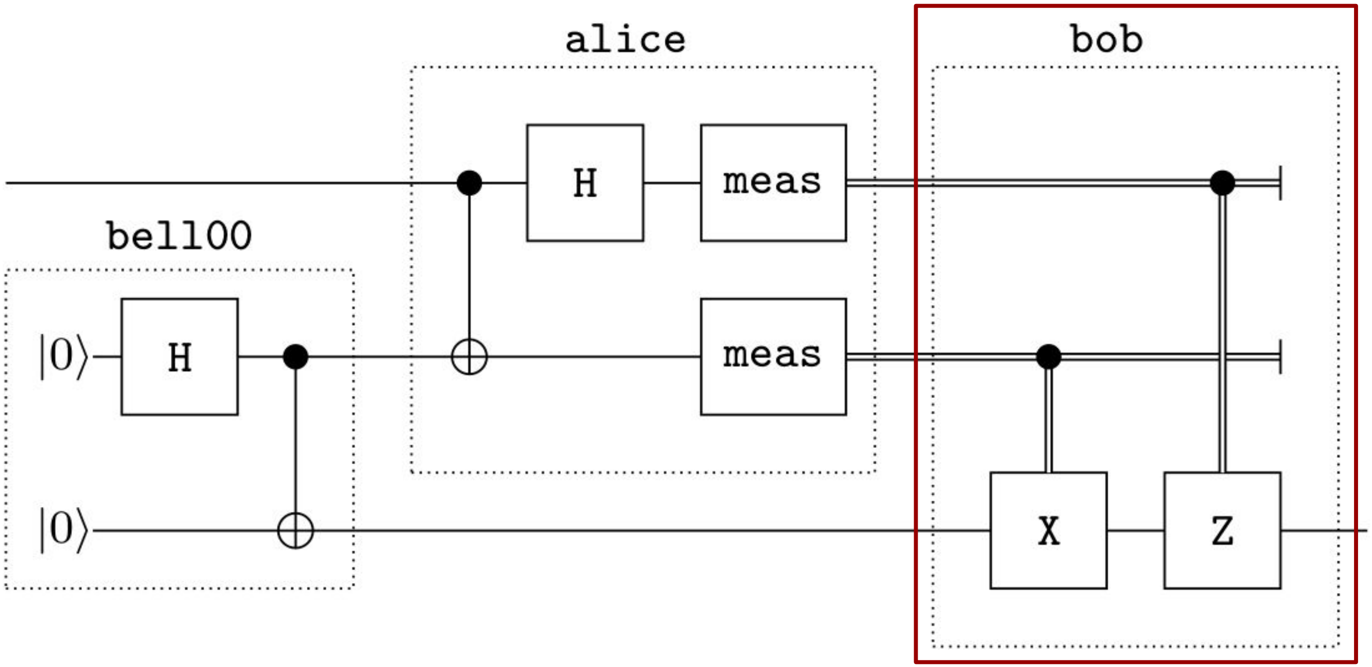


```
inPar (c : Circ(W1,W2)) (c’ : Circ(W1’,W2’))
 : Circ(W1⊗W1’, W2⊗W2’) =
 box (w1,w1’) =>
  w2  <- unbox c  w1;
  w2’ <- unbox c’ w1’;
  output (w2,w2’)
```



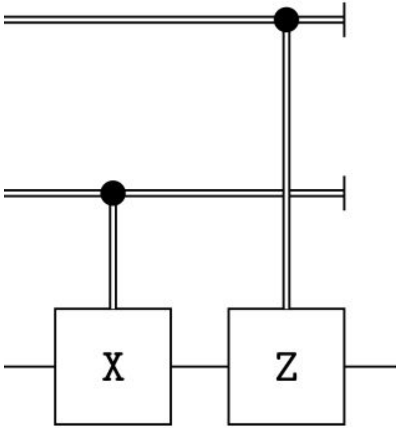
In both cases the type system guarantees that the wire types match

Dynamic Lifting - example with Quantum Teleportation



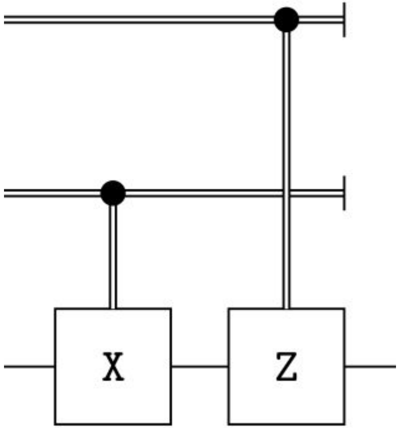
Bob with Dynamic Lifting

```
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 with Dynamic Lifting

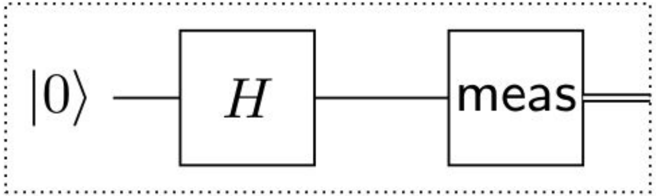
```
bob : Circ(bit⊗bit⊗qubit, qubit) =
  box (x,y,b) =>
    (y,b) <- gate (bit-control X) (y,b);
    (x,b) <- gate (bit-control Z) (x,b);
    ()    <- gate discard y;
    ()    <- gate discard x;
    output b
```



```
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 $W ::= 1 \mid \mathsf{bit} \mid \mathsf{qubit} \mid W_1 \otimes W_2$

Gates have input and output wire type, and we assume

- If a unitary $u \in \mathcal{G}(W, W)$ then

$$u^\dagger \in \mathcal{G}(W, W)$$

$$\mathsf{control}\ u \in \mathcal{G}(\mathsf{qubit} \otimes W, \mathsf{qubit} \otimes W)$$

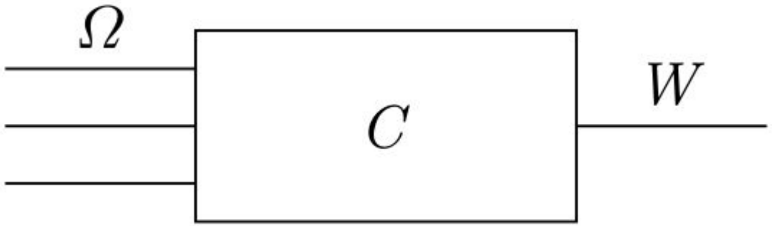
$$\mathsf{bit\text{-}control}\ u \in \mathcal{G}(\mathsf{bit} \otimes W, \mathsf{bit} \otimes W)$$

- Initialization $\mathsf{new0}, \mathsf{new1} \in \mathcal{G}(1, \mathsf{bit}), \mathsf{init0}, \mathsf{init1} \in \mathcal{G}(1, \mathsf{qubit})$
- Measurement and discard $\mathsf{meas} \in \mathcal{G}(\mathsf{qubit}, \mathsf{bit})$ $\mathsf{discard} \in \mathcal{G}(\mathsf{bit}, 1)$

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 **host** variables with host types
- $\Omega = w_1 : W_1, \ldots, w_n : W_n$ context of **wire** variables with wire types
- C a circuit
- W the output wire type



Auxiliary Judgement

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

$$\frac{}{\cdot \Rightarrow () : 1} \qquad \frac{}{w : W \Rightarrow w : W}$$

$$\frac{\Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2}{\Omega_1, \Omega_2 \Rightarrow (p_1, p_2) : W_1 \otimes W_2}$$

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)

$$|\mathsf{bit}| = \mathsf{Bool} \qquad |1| = \mathsf{Unit}$$

$$|\mathsf{qubit}| = \mathsf{Bool} \qquad |W_1 \otimes W_2| = |W_1| \times |W_2|$$

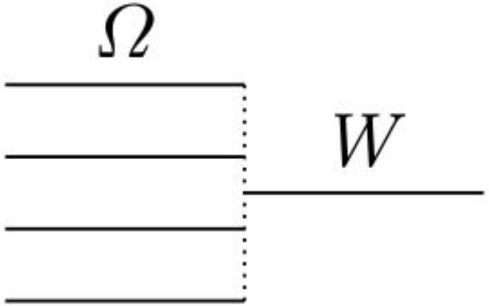
- a type for circuits with typed input and output $\mathsf{Circ}(W_1, W_2)$

Typing Rules for Qwire

Output

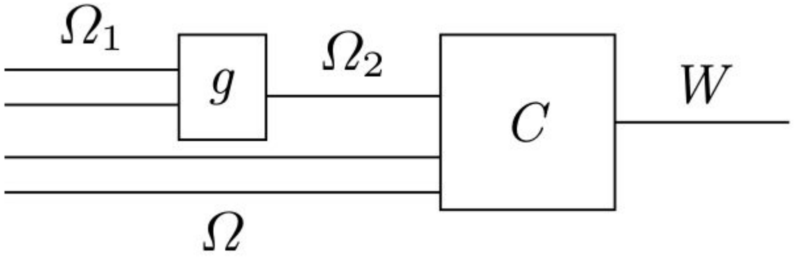
- Build a pattern of its input wires

$$\frac{\Omega \Rightarrow p : W}{\Gamma; \Omega \vdash \mathsf{output}\ p : W}\ \text{OUTPUT}$$



Typing Rules for Qwire

Gates



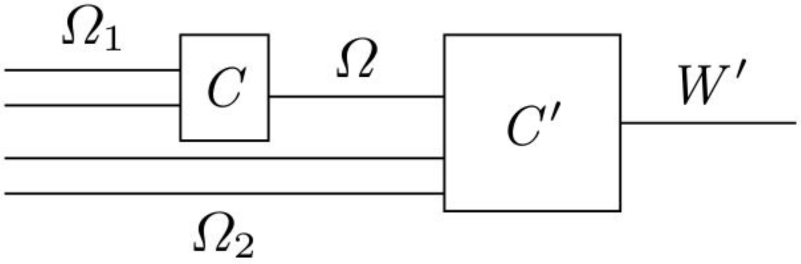
- Are applied to a pattern of wires
- The output defines another pattern
- Unused wires and output wires are used in the continuation

$$\frac{\begin{array}{l} g \in \mathcal{G}(W_1, W_2) \\ \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \Gamma; \Omega_2, \Omega \vdash C : W \end{array}}{\Gamma; \Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; C : W} \ \text{GATE}$$

Typing Rules for Qwire

Composition

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

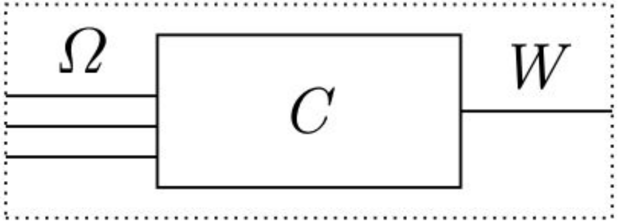


$$\frac{\Gamma; \Omega_1 \vdash C : W \quad \Omega \Rightarrow p : W \quad \Gamma; \Omega, \Omega_2 \vdash C' : W'}{\Gamma; \Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'} \text{ COMPOSE}$$

Typing Rules for Qwire

Boxing

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

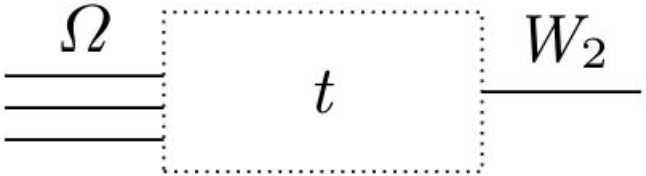


$$\frac{\Omega \Rightarrow p : W_1 \quad \Gamma ; \Omega \vdash C : W_2}{\Gamma \vdash \mathsf{box}\,(p : W_1) \Rightarrow C : \mathsf{Circ}(W_1, W_2)} \; \text{BOX}$$

Typing Rules for Qwire

Unboxing

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



$$\frac{\Gamma \vdash t : \mathsf{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1}{\Gamma; \Omega \vdash \mathsf{unbox}\; t\; p : W_2} \text{ UNBOX}$$

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

$$
\begin{aligned}
|\mathsf{bit}| &= \mathsf{Bool} \\
|\mathsf{qubit}| &= \mathsf{Bool} \\
|1| &= \mathsf{Unit} \\
|W_1 \otimes W_2| &= |W_1| \times |W_2|
\end{aligned}
$$

$$\frac{\Gamma; \cdot \vdash C : W}{\Gamma \vdash \mathsf{run}\ C : |W|} \ \mathrm{RUN}$$

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

$$\frac{\Omega \Rightarrow p : W \qquad \Gamma, x : |W|; \Omega' \vdash C : W'}{\Gamma; \Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W'}\ \mathrm{LIFT}$$

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

$$C \Longrightarrow C'$$

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

$$\cdot ; \mathcal{Q} \vdash C : W \qquad \text{where} \qquad \mathcal{Q} ::= \cdot \mid \mathcal{Q}, w : \mathsf{bit} \mid \mathcal{Q}, w : \mathsf{qubit}$$

$$N ::= \mathsf{output}\ p \mid p_2 \leftarrow \mathsf{gate}\ g\ p_1 ; N \mid x \Leftarrow \mathsf{lift}\ p ; C$$

Operational Semantics - ingredients

- Pattern generalization $p' \preccurlyeq p$ (e.g. $p \preccurlyeq w$)
- 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 $w : \mathsf{bit} \otimes \mathsf{bit} \Rightarrow w : \mathsf{bit} \otimes \mathsf{bit}$

    to $w' : \mathsf{bit}, w : \mathsf{bit} \Rightarrow (\, w', \, w\,) : \mathsf{bit} \otimes \mathsf{bit}$
- Host terms evaluation $t \longrightarrow t'$ is the union of
  - Host language alone $\longrightarrow_{\mathrm{H}}$
  - Boxed circuits $\longrightarrow_{b}$

Operational Semantics

Box

- Concretizes the pattern first
- Then normalizes the circuit

$$\frac{p \text{ is concrete for } W \qquad C \Longrightarrow C'}{\mathsf{box}\,(p : W) \Rightarrow C \longrightarrow_b \mathsf{box}\,(p : W) \Rightarrow C'}\ \textsc{struct}$$

$$\frac{p' \prec p \qquad p' \text{ is concrete for } W}{(\mathsf{box}\,(p : W) \Rightarrow C) \longrightarrow_b (\mathsf{box}\ p' \Rightarrow C\,\{p'/p\})}\ \eta$$

Operational Semantics

Unbox

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

$$\frac{t \longrightarrow t'}{\mathsf{unbox}\ t\ p \Longrightarrow \mathsf{unbox}\ t'\ p}\ \textsc{Struct}$$

$$\frac{}{\mathsf{unbox}\ (\mathsf{box}\ (p : W) \Rightarrow N)\ p' \Longrightarrow N\ \{p'/p\}}\ \beta$$

Operational Semantics

Gate

- Concretizes the pattern first
- Then proceeds with the continuation

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad p_2 \text{ is concrete for } W_2 \quad C \Longrightarrow C'}{p_2 \leftarrow \textsf{gate}\ g\ p_1;\ C \Longrightarrow p_2 \leftarrow \textsf{gate}\ g\ p_1;\ C'}\ \textsc{STRUCT}$$

$$\frac{g \in \mathcal{G}(W_1, W_2) \quad p_2' \prec p_2 \quad p_2' \text{ is concrete for } W_2}{p_2 \leftarrow \textsf{gate}\ g\ p_1;\ C \Longrightarrow p_2' \leftarrow \textsf{gate}\ g\ p_1;\ C\ \{p_2'/p_2\}}\ \eta$$

Operational Semantics

Composition

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

$$\frac{C_1 \Longrightarrow C_1'}{p \leftarrow C_1; C_2 \Longrightarrow p \leftarrow C_1'; C_2} \quad \text{STRUCT}$$

$$\frac{}{p \leftarrow \mathsf{output}\ p'; C \Longrightarrow C\ \{p'/p\}} \quad \beta$$

Operational Semantics

Composition

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

$$\frac{}{p \leftarrow (p_2 \leftarrow \mathsf{gate}\ g\ p_1\, ;\, N);\, C \Longrightarrow p_2 \leftarrow \mathsf{gate}\ g\ p_1\, ;\, p \leftarrow N;\, C}\ \textbf{CC}$$

$$\frac{}{p' \leftarrow (x \Leftarrow \mathsf{lift}\ p;\, C');\, C \Longrightarrow x \Leftarrow \mathsf{lift}\ p;\, p' \leftarrow C';\, C}\ \textbf{CC}$$

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 $\longrightarrow_{\mathrm{H}}$ 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

$$[\cdot] = \mathcal{H}_1 \qquad [w : W] = [W] \qquad [\Omega_1, \Omega_2] = [\Omega_1] \otimes [\Omega_2]$$

with:

$$[\mathsf{bit}] = \mathcal{H}_2 \qquad [1] = \mathcal{H}_1$$

$$[\mathsf{qubit}] = \mathcal{H}_2 \qquad [W_1 \otimes W_2] = [W_1] \otimes [W_2]$$

Denotational Semantics

- Semantics of values is as expected

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

$$
\begin{aligned}
[\![* : \mathsf{Unit}]\!] &= |*\rangle \\
[\![\mathsf{false} : \mathsf{Bool}]\!] &= |0\rangle \\
[\![\mathsf{true} : \mathsf{Bool}]\!] &= |1\rangle \\
[\![(v_1, v_2) : |W_1| \times |W_2|]\!] &= [\![v_1 : |W_1|]\!] \otimes [\![v_2 : |W_2|]\!]
\end{aligned}
$$

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

$$\begin{aligned} [\![\mathsf{new0}]\!], [\![\mathsf{init0}]\!] &= (|0\rangle\langle 0|)^* \\ [\![\mathsf{new1}]\!], [\![\mathsf{init1}]\!] &= (|1\rangle\langle 1|)^* \\ [\![\mathsf{meas}]\!] &= (|0\rangle\langle 0|)^* + (|1\rangle\langle 1|)^* \\ [\![\mathsf{discard}]\!] &= \langle 0|^* + \langle 1|^* \end{aligned}$$

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

Denotational Semantics

$$\frac{\Omega \Rightarrow p : W}{\cdot;\Omega \vdash \mathsf{output}\ p : W}$$

$$[\![\Omega \vdash \mathsf{output}\ p : W]\!] = \mathbf{I}^*$$

$$\frac{\cdot;\Omega' \vdash C : W \quad \pi : \Omega \equiv \Omega'}{\cdot;\Omega \vdash C : W}$$

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

$$\frac{\cdot \vdash t : \mathsf{Circ}(W_1, W_2) \quad \Omega \Rightarrow p : W_1}{\cdot;\Omega \vdash \mathsf{unbox}\ t\ p : W_2}$$

$$[\![\Omega \vdash \mathsf{unbox}\ t\ p : W']\!] = [\![t : \mathsf{Circ}(W, W')]\!]$$

$$\frac{\begin{array}{l} g \in \mathcal{G}(W_1, W_2) \\ \Omega_1 \Rightarrow p_1 : W_1 \quad \Omega_2 \Rightarrow p_2 : W_2 \quad \cdot;\Omega_2, \Omega \vdash C : W \end{array}}{\cdot;\Omega_1, \Omega \vdash p_2 \leftarrow \mathsf{gate}\ g\ p_1; C : W}$$

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

$$\frac{\Omega \Rightarrow p : W \quad x : |W|; \Omega' \vdash C : W'}{\cdot;\Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W'}$$

$$[\![\Omega, \Omega' \vdash x \Leftarrow \mathsf{lift}\ p; C : W']\!] = \sum_{\cdot \vdash v : |W|} [\![\Omega' \vdash C\{v/x\} : W']\!] \circ ([v : |W|]^\dagger \otimes \mathbf{I})^*$$

$$\frac{\cdot;\Omega_1 \vdash C : W \quad \Omega_0 \Rightarrow p : W \quad \cdot;\Omega_0, \Omega_2 \vdash C' : W'}{\cdot;\Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W'}$$

$$[\![\Omega_1, \Omega_2 \vdash p \leftarrow C; C' : W']\!] = [\![\Omega_0, \Omega_2 \vdash C' : W']\!] \circ ([\![\Omega_1 \vdash C : W]\!] \otimes \mathbf{I}^*)$$

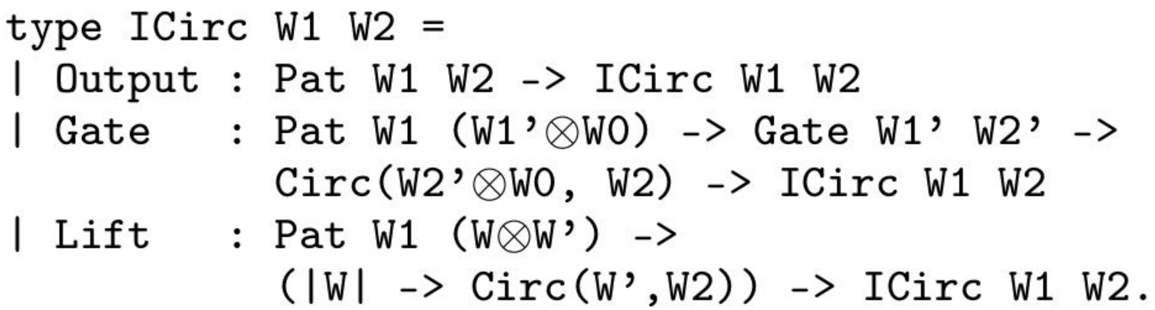
Using Qwires

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 $\mathsf{Circ}(W_1, W_2)$

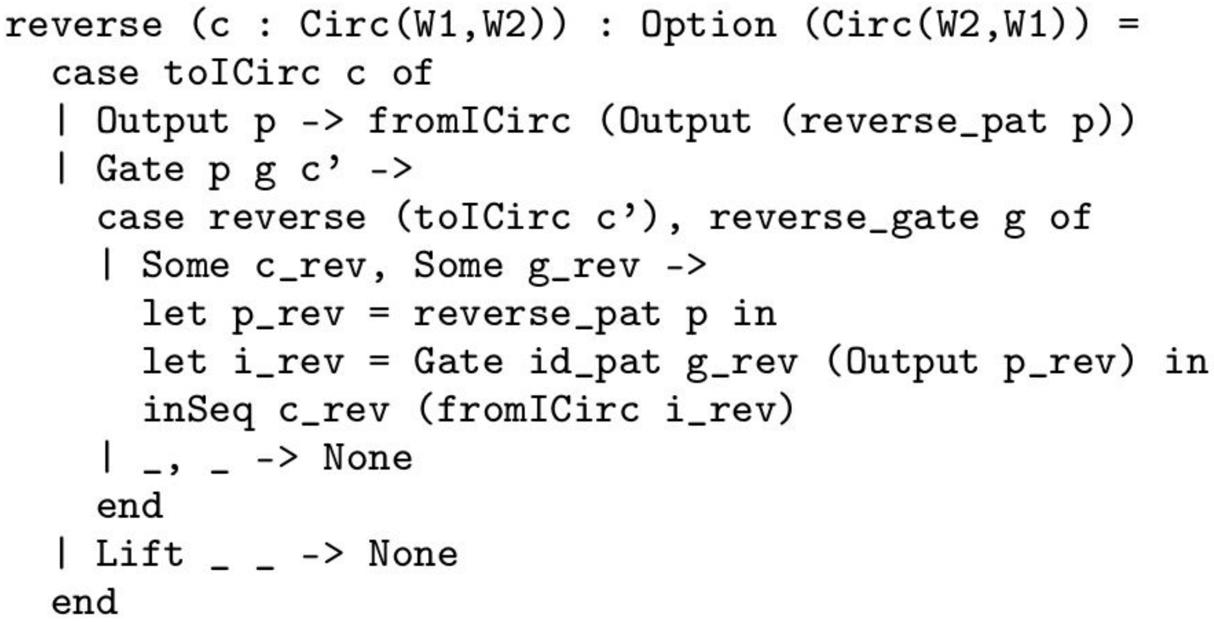


```
type ICirc W1 W2 =
| Output : Pat W1 W2 -> ICirc W1 W2
| Gate   : Pat W1 (W1'⊗W0) -> Gate W1' W2' ->
           Circ(W2'⊗W0, W2) -> ICirc W1 W2
| Lift   : Pat W1 (W⊗W') ->
           (|W| -> Circ(W',W2)) -> ICirc W1 W2.
```

- 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



```
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

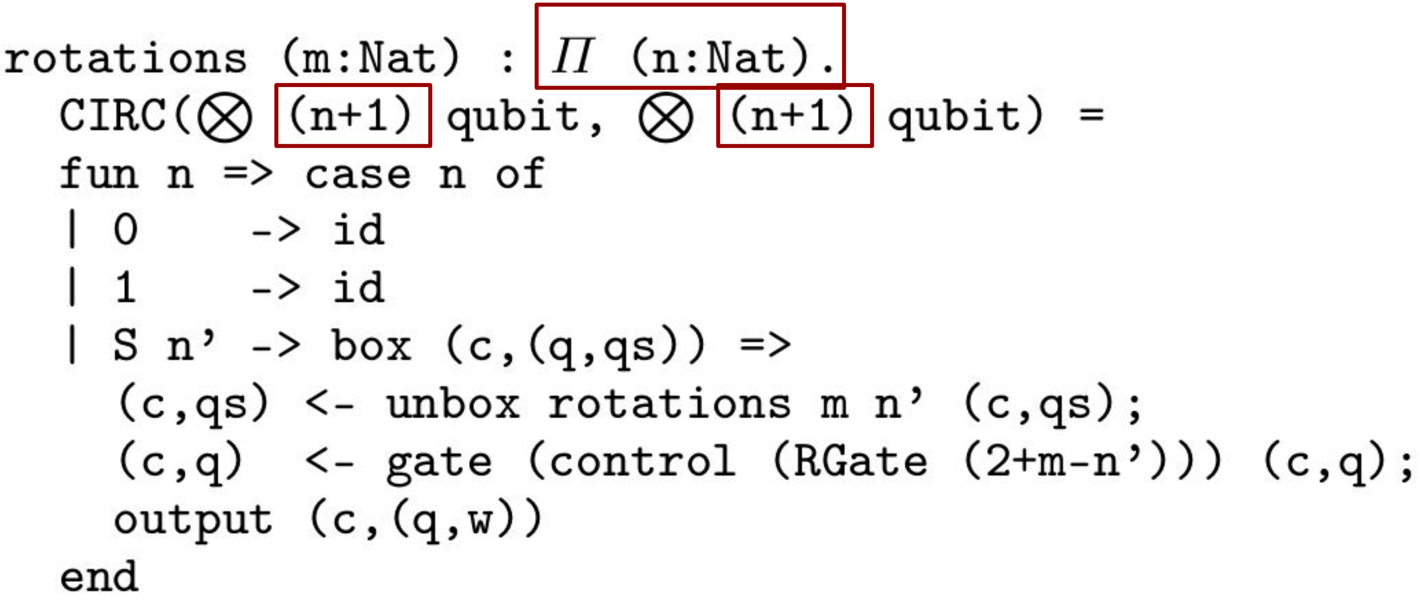
- We would like to have functions that generate circuits with a number of wires that depends on the input, i.e. dependent types

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) : Π (n:Nat).
  CIRC(⊗ (n+1) qubit, ⊗ (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**

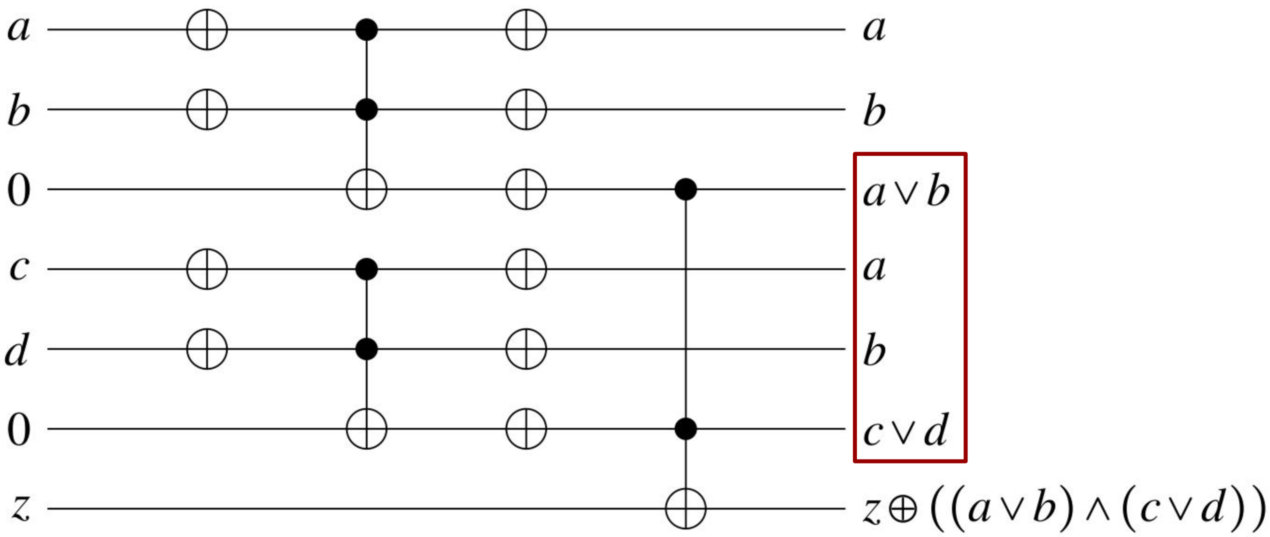
- Reversible logic circuits require ancillae
- After usage, we want to discard them
- We must be sure of their state to do that
  - Measuring an entangled qubit affects the result!

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



ReQwire: Reasoning about Reversible Circuits

We add assertion gates that discard a qubit in the given state

```
g := U | init_0 | init_1 | meas | discard | assert_0 | 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**

ReQwire: Reasoning about Reversible Circuits

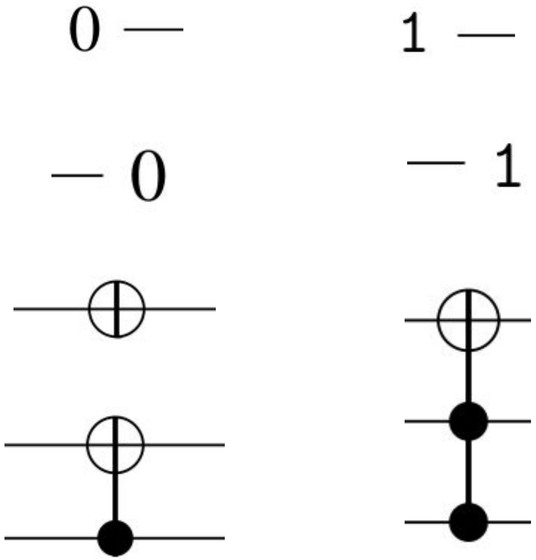
Based on the denotational precise definition of validity

- 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
    - (`init_`b at **i**) ;; **c** ;; (`assert_`b at **i**) is source symmetric

ReQwire: Reasoning about Reversible Circuits

Compiling Oracles

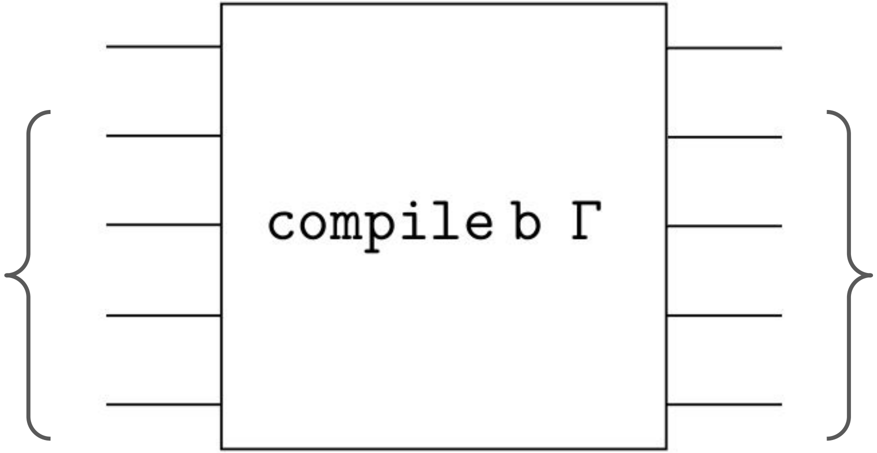
- Given a boolean expression

$$b ::= x \mid t \mid f \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \oplus b_2$$

- and a map Γ 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



Target

Source wires

$\texttt{compile}\ b\ \Gamma$

$[\![b]\!] \oplus$ Target

Unchanged source wires

ReQwire: Reasoning about Reversible Circuits

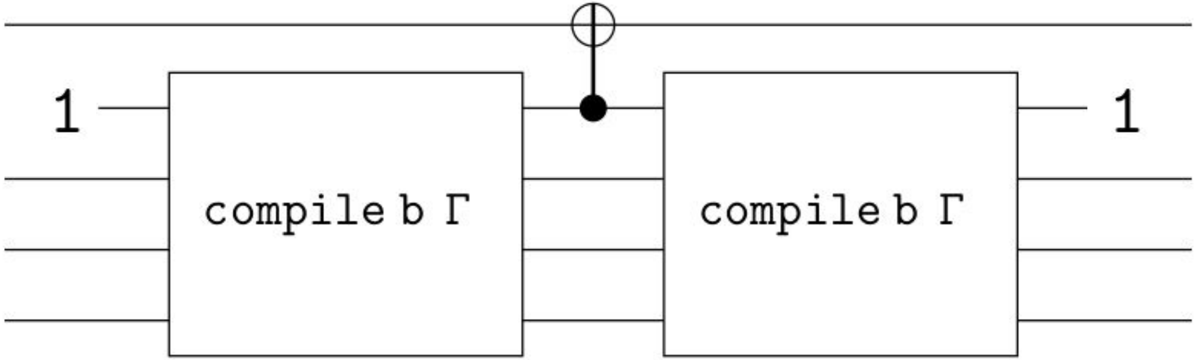
Compiling a variable, true and false

$$b ::= \boxed{x \mid t \mid f \mid} \neg b \mid b_1 \wedge b_2 \mid b_1 \oplus b_2$$

ReQwire: Reasoning about Reversible Circuits

Compiling a negation

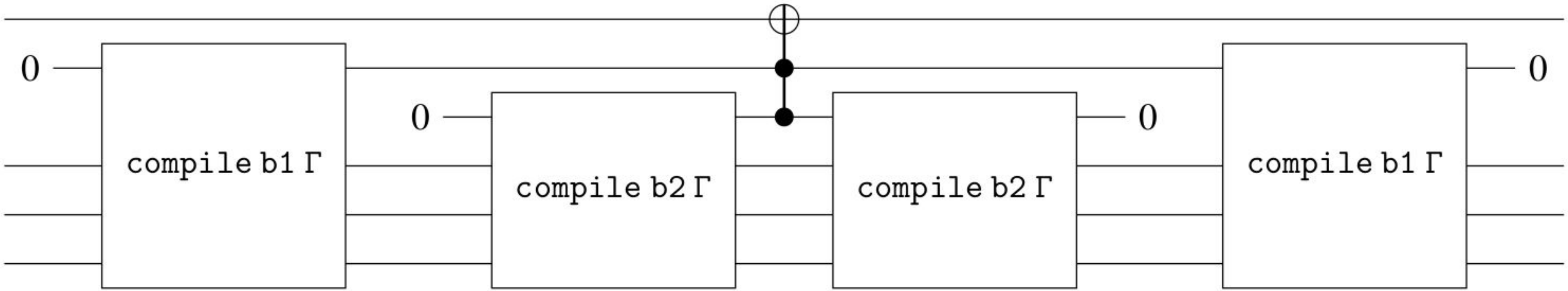
$$b ::= x \mid t \mid f \mid \boxed{\neg b} \mid b_1 \wedge b_2 \mid b_1 \oplus b_2$$



ReQwire: Reasoning about Reversible Circuits

Compiling a conjunction

$$b ::= x \mid t \mid f \mid \neg b \mid \boxed{b_1 \wedge b_2} \mid b_1 \oplus b_2$$

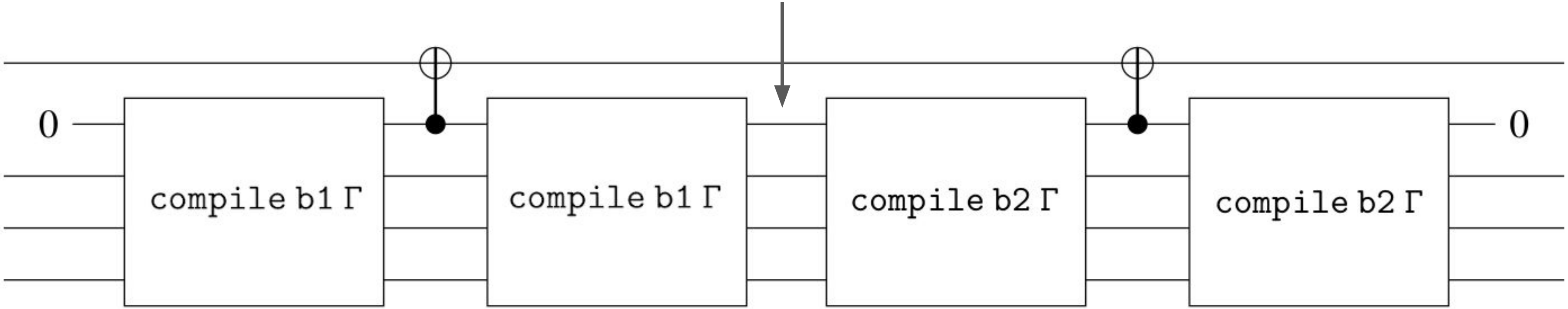


ReQwire: Reasoning about Reversible Circuits

Compiling an exclusive disjunction

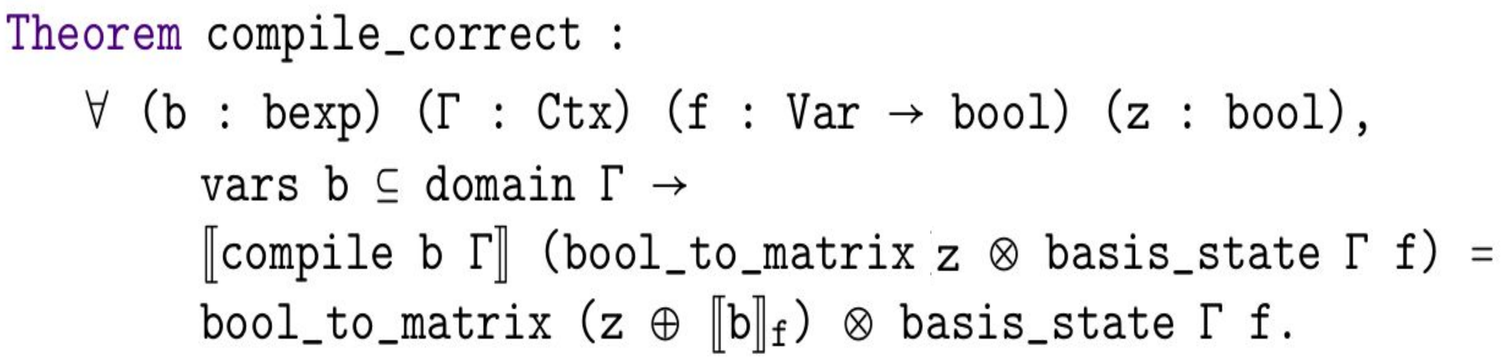
$$b ::= x \mid t \mid f \mid \neg b \mid b_1 \wedge b_2 \mid \boxed{b_1 \oplus b_2}$$

here we
have 0



ReQwire: Reasoning about Reversible Circuits

Correctness of the compilation (in Coq)



```
Theorem compile_correct :
    ∀ (b : bexp) (Γ : Ctx) (f : Var → bool) (z : bool),
        vars b ⊆ domain Γ →
        ⟦compile b Γ⟧ (bool_to_matrix z ⊗ basis_state Γ f) =
        bool_to_matrix (z ⊕ ⟦b⟧_f) ⊗ basis_state Γ 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

- **QWIRE: a core language for quantum circuits**, by *J. Paykin, R. Rand and S. Zdancewic*. POPL 2017

- **ReQWIRE: Reasoning about Reversible Quantum Circuits**, by *R. Rand, J. Paykin, D. Lee, S. Zdancewic*. QPL 2018