Tracking Errors through Types in Quantum Programs

Kesha Hietala, *Robert Rand*, Michael Hicks

Programming Languages for Quantum Computing

PLanQC 2020

Overview

- Errors, particularly those introduced by gate application, will be prevalent in near-term quantum machines, so need to be taken into account when writing programs.
- It might be useful to have a notion of errors at the *programming language level*
- We have designed a simple type system to track errors in quantum programs, implemented in **QWIRE**

QWIRE

QWIRE

- A small but expressive quantum circuit language

QWIRE

- A small but expressive quantum circuit language
- Embedded in the Coq proof assistant

QWIRE

- A small but expressive quantum circuit language
- Embedded in the Coq proof assistant

QWIRE

- A small but expressive quantum circuit language
- Embedded in the Coq proof assistant
- A linear type system for enforcing no-cloning

QWIRE

- A small but expressive quantum circuit language
- Embedded in the Coq proof assistant
- A linear type system for enforcing no-cloning
- A denotational semantics in terms of density matrices

QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```

QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```

QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```

QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```

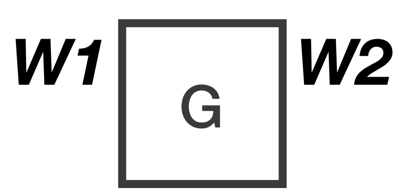
QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```

p : W

QWIRE

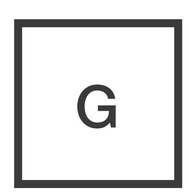
```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```



QWIRE

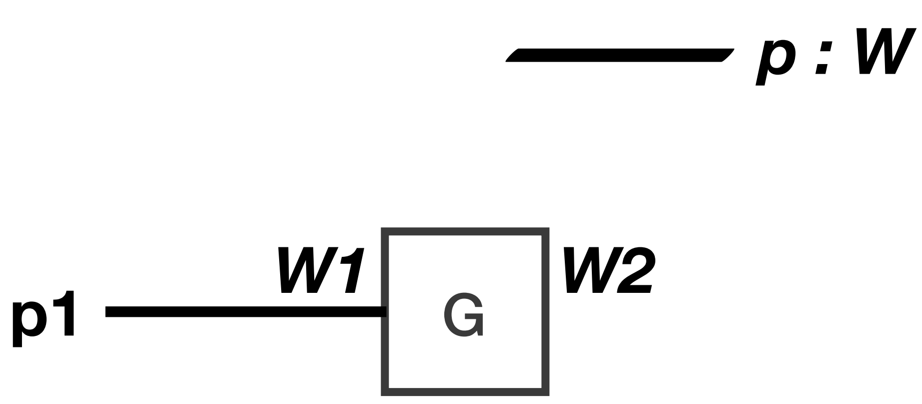
```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```

p : W

W1  *W2*

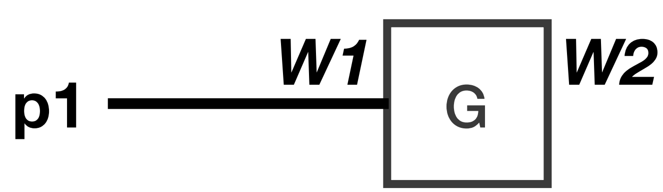
QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```



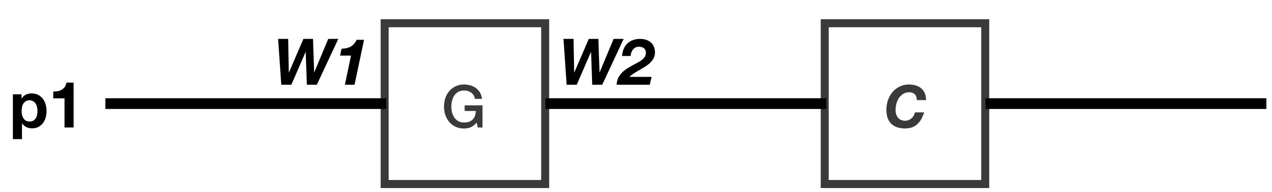
QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```



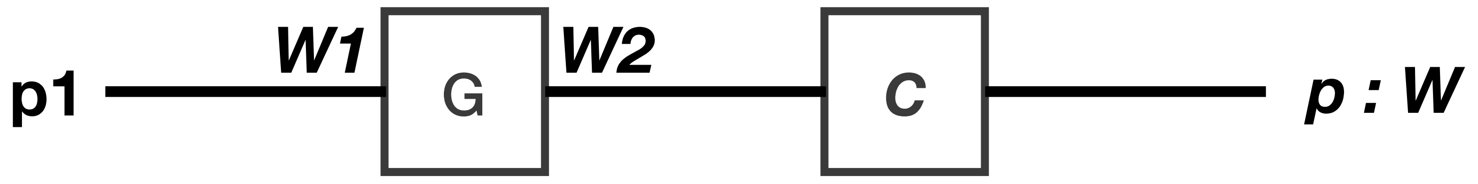
QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```



QWIRE

```
Inductive Circuit (W : WType) : Type :=
| output : Pat W -> Circuit W
| gate   : Gate W1 W2 -> Pat W1 ->
           (Pat W2 -> Circuit W) -> Circuit W
| lift   : Pat Bit -> (bool -> Circuit W) ->
           Circuit W.
```



Wire Types

```
Inductive WType := Qubit | Bit | One | W ⊗ W.
```

Wire Types

```
Inductive WType := Qubit | Bit | One | W ⊗ W.
```

```
Inductive Gate : WType -> WType -> Set :=
 | U       : Unitary W -> Gate W W
 | init0   : Gate One Qubit
 | new0    : Gate One Bit
 | meas    : Gate Qubit Bit
 | discard : Gate Bit One.
```

Wire Types

```
Inductive WType := Qubit | Bit | One | W ⊗ W.
```

```
Inductive Gate : WType -> WType -> Set :=
  | U       : Unitary W -> Gate W W
  | init0   : Gate One Qubit
  | new0    : Gate One Bit
  | meas    : Gate Qubit Bit
  | discard : Gate Bit One.
```

What *Kind* of Qubit?

What *Kind* of Qubit?

- *Physical qubits:* Individual machine qubits, prone to error/degradation.

What *Kind* of Qubit?

- *Physical qubits:* Individual machine qubits, prone to error/degradation.
- *Logical qubits:* 100 - 1000s of physical qubits, assumed to be highly error tolerant and regularly corrected.

What *Kind* of Qubit?

- *Physical qubits:* Individual machine qubits, prone to error/degradation.
- *Logical qubits:* 100 - 1000s of physical qubits, assumed to be highly error tolerant and regularly corrected.
- *Simple Encoded Qubits:* Some small number of physical qubits (say, 9) correspond to a qubit.

What *Kind* of Qubit?

- *Physical qubits:* Individual machine qubits, prone to error/degradation.
- *Logical qubits:* 100 - 1000s of physical qubits, assumed to be highly error tolerant and regularly corrected.
- *Simple Encoded Qubits:* Some small number of physical qubits (say, 9) correspond to a qubit.

$$|0\rangle_E = (\,|000\rangle + |111\rangle) \otimes (\,|000\rangle + |111\rangle) \otimes (\,|000\rangle + |111\rangle)$$

$$|1\rangle_E = (\,|000\rangle - |111\rangle) \otimes (\,|000\rangle - |111\rangle) \otimes (\,|000\rangle - |111\rangle)$$

Adding Errors

```
Inductive WType := Qubit k | Bit | One | W ⊗ W.
```

```
Inductive Gate :=
  | U | init0 | init1 | meas | discard | EC.
```

Adding Errors

```
Inductive WType := Qubit k | Bit | One | W ⊗ W.
```

```
Inductive Gate :=
  | U | init0 | init1 | meas | discard | EC.
```

Adding Errors

```
Inductive WType := Qubit k | Bit | One | W ⊗ W.
```

```
Inductive Gate :=
  | U | init0 | init1 | meas | discard | EC.
```

Error Prone Gates

```
Inductive Gate : nat -> WType -> WType -> Set :=
 | U       : (U : Unitary k W) ->
             Gate k W (map W (k + sum_err W))
 | init0   : Gate ϵ_i One (Qubit ϵ_i)
 | new0    : Gate 0 One Bit
 | meas    : Gate ϵ_m (Qubit n) Bit
 | discard : Gate 0 Bit One
 | EC      : Gate ϵ_e (Qubit n) (Qubit 0).
```

Error Prone Gates

error term

```
Inductive Gate : nat -> WType -> WType -> Set :=
  | U       : (U : Unitary k W) ->
              Gate k W (map W (k + sum_err W))
  | init0   : Gate ϵᵢ One (Qubit ϵᵢ)
  | new0    : Gate 0 One Bit
  | meas    : Gate ϵₘ (Qubit n) Bit
  | discard : Gate 0 Bit One
  | EC      : Gate ϵₑ (Qubit n) (Qubit 0).
```

Error Prone Gates

error term

```
Inductive Gate : nat -> WType -> WType -> Set :=
  | U       : (U : Unitary k W) ->
              Gate k W (map W (k + sum_err W))
  | init0   : Gate ϵi One (Qubit ϵi)
  | new0    : Gate 0 One Bit
  | meas    : Gate ϵm (Qubit n) Bit
  | discard : Gate 0 Bit One
  | EC      : Gate ϵe (Qubit n) (Qubit 0).
```

Error Prone Gates

error term

```
Inductive Gate : nat -> WType -> WType -> Set :=
  | U       : (U : Unitary k W) ->
              Gate k W (map W (k + sum_err W))
  | init0   : Gate ϵ_i One (Qubit ϵ_i)
  | new0    : Gate 0 One Bit
  | meas    : Gate ϵ_m (Qubit n) Bit
  | discard : Gate 0 Bit One
  | EC      : Gate ϵ_e (Qubit n) (Qubit 0).
```

Error Prone Gates

error term

```
Inductive Gate : nat -> WType -> WType -> Set :=
  | U       : (U : Unitary k W) ->
              Gate k W (map W (k + sum_err W))
  | init0   : Gate ϵ_i One (Qubit ϵ_i)
  | new0    : Gate 0 One Bit
  | meas    : Gate ϵ_m (Qubit n) Bit
  | discard : Gate 0 Bit One
  | EC      : Gate ϵ_e (Qubit n) (Qubit 0).
```

Fault Tolerance

Gottesman (2009): An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation

Fault Tolerance

- We say that an operation O is fault tolerant if, provided there are not too many errors on the input, (i) O does not introduce too many errors and (ii) O performs the intended operation.

Gottesman (2009): An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation

Fault Tolerance

- We say that an operation O is fault tolerant if, provided there are not too many errors on the input, (i) O does not introduce too many errors and (ii) O performs the intended operation.

- We assume the existence of an *ideal decoder* that corrects up to t errors.

Gottesman (2009): An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation

Fault Tolerance

- We say that an operation O is fault tolerant if, provided there are not too many errors on the input, (i) O does not introduce too many errors and (ii) O performs the intended operation.
- We assume the existence of an *ideal decoder* that corrects up to *t* errors.
- Unitary operation *U* is fault tolerant if $\sum \epsilon_{in} + \epsilon_{gate} \leq t$ implies

Gottesman (2009): An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation

Fault Tolerance

- We say that an operation O is fault tolerant if, provided there are not too many errors on the input, (i) O does not introduce too many errors and (ii) O performs the intended operation.

- We assume the existence of an *ideal decoder* that corrects up to *t* errors.

- Unitary operation *U* is fault tolerant if $\sum \epsilon_{in} + \epsilon_{gate} \leq t$ implies
  1. $\epsilon_{out} \leq \sum \epsilon_{in} + \epsilon_{gate}$

Gottesman (2009): An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation

Fault Tolerance

- We say that an operation O is fault tolerant if, provided there are not too many errors on the input, (i) O does not introduce too many errors and (ii) O performs the intended operation.

- We assume the existence of an *ideal decoder* that corrects up to *t* errors.

- Unitary operation *U* is fault tolerant if $\sum \epsilon_{in} + \epsilon_{gate} \leq t$ implies
  1. $\epsilon_{out} \leq \sum \epsilon_{in} + \epsilon_{gate}$
  2. $decode(U; |\psi\rangle) = U_{ideal};\ decode(\ |\psi\rangle)$

Gottesman (2009): An Introduction to Quantum Error Correction and Fault-Tolerant Quantum Computation

Error Correction Gates

Error Correction Gates

- Error correction gate *EC* is fault tolerant if

Error Correction Gates

- Error correction gate *EC* is fault tolerant if

  1. $\epsilon_{out} \leq t$

Error Correction Gates

- Error correction gate *EC* is fault tolerant if

    1. $\epsilon_{out} \leq t$

    2. $\epsilon_{in} + \epsilon_{gate} \leq t$ implies
    $decode(EC; |\psi\rangle) = decode(|\psi\rangle)$

Checking Fault Tolerance

Checking Fault Tolerance

- We check these using QWIRE’s type system

Checking Fault Tolerance

- We check these using QWIRE's type system
- We benefit from *linearity* — we cannot introduce an error *onto* an existing qubit, but only produce a new qubit with some number of errors

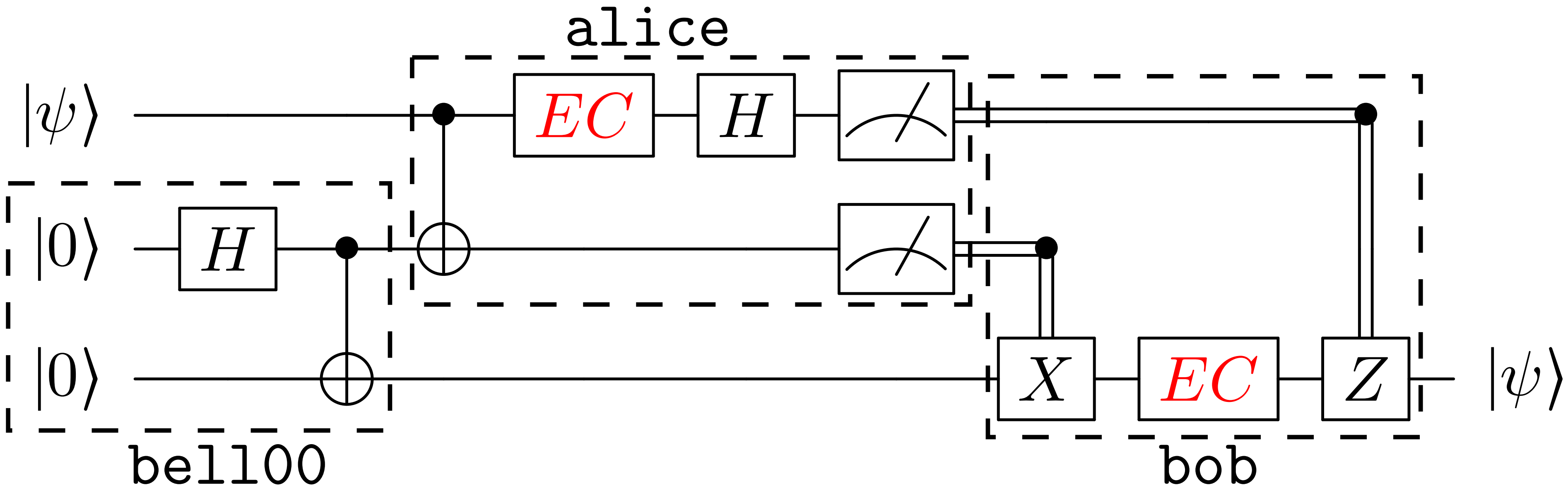
Checking Fault Tolerance

- We check these using QWIRE's type system
- We benefit from *linearity* — we cannot introduce an error *onto* an existing qubit, but only produce a new qubit with some number of errors
- However, *checking* fault tolerance is orthogonal to checking linearity.

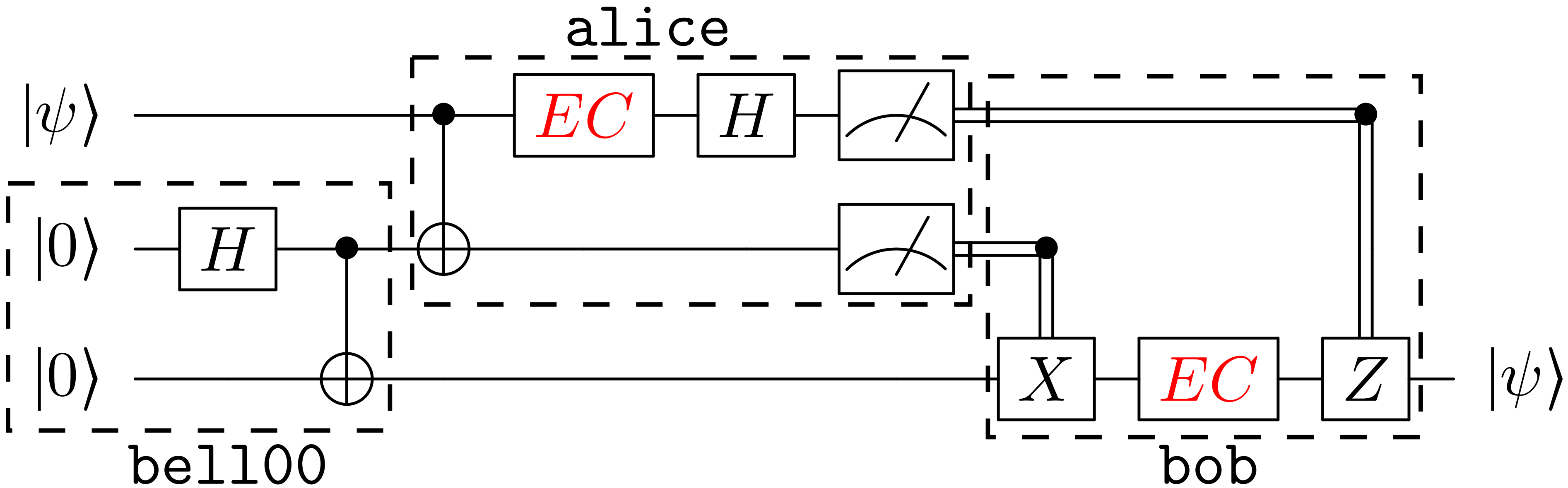
Typechecking

```
Inductive Types_Circuit :  OCtx -> nat -> Circuit W -> Set :=
 | types_gate : forall {Γ Γ1 Γ1' w1 w2 w}
                (f : Pat w2 -> Circuit w)
                (k t : nat) {p1 : Pat w1} {g : Gate k w1 w2},
                Γ1 ⊢ p1 :Pat ->
                k + sum_err W1 <= t ->
               (forall Γ2 Γ2' (p2 : Pat w2)
                  {pf2 : Γ2' == Γ2 · Γ},
                  Γ2 ⊢ p2 :Pat -> Types_Circuit Γ2' t (f p2)) ->
               forall {pf1 : Γ1' == Γ1 · Γ},
               Types_Circuit Γ1' t (gate g p1 f).
```

Typechecking

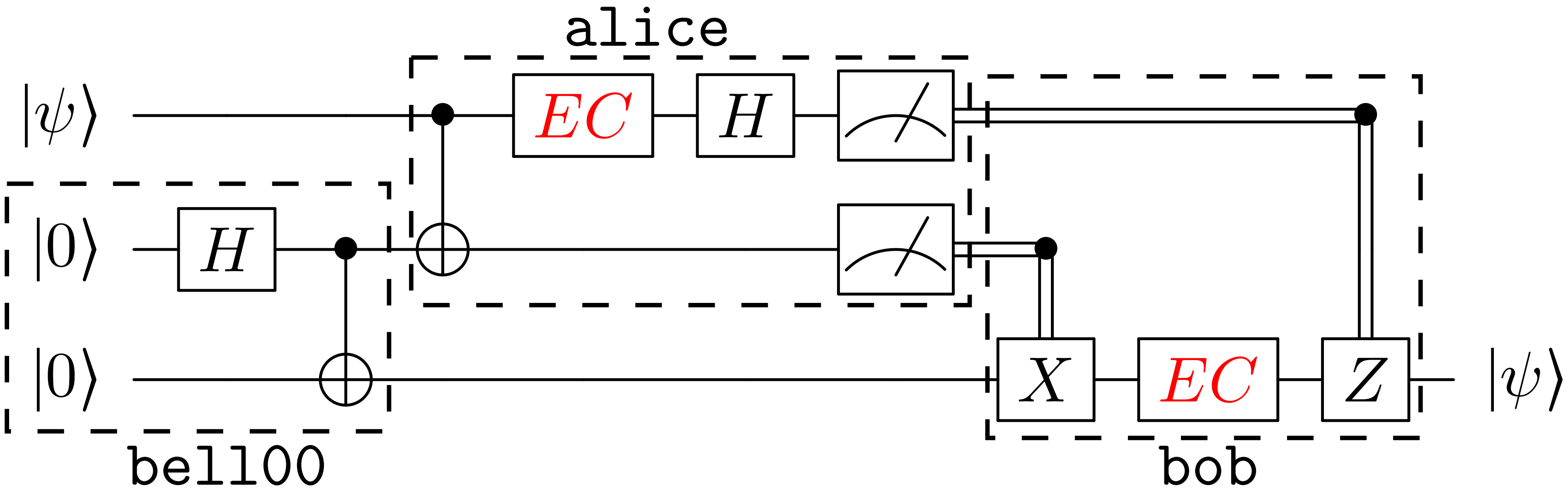


Typechecking



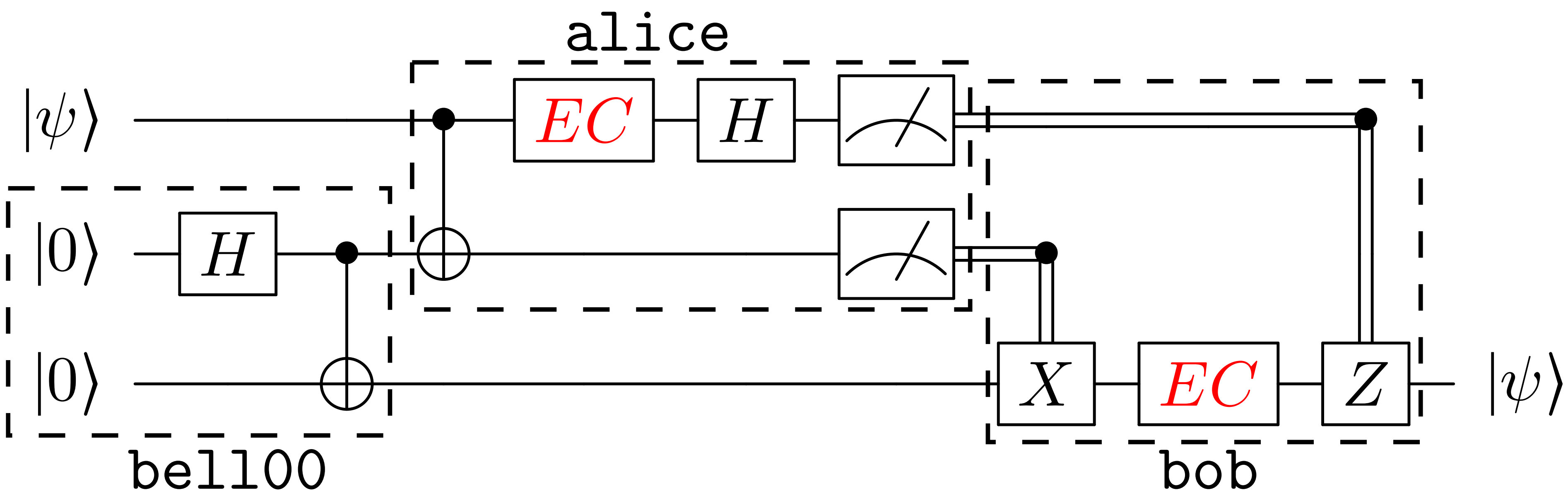

```
Definition bell : Box One (Qubit 2 ⊗ Qubit 2) := …
```

Typechecking



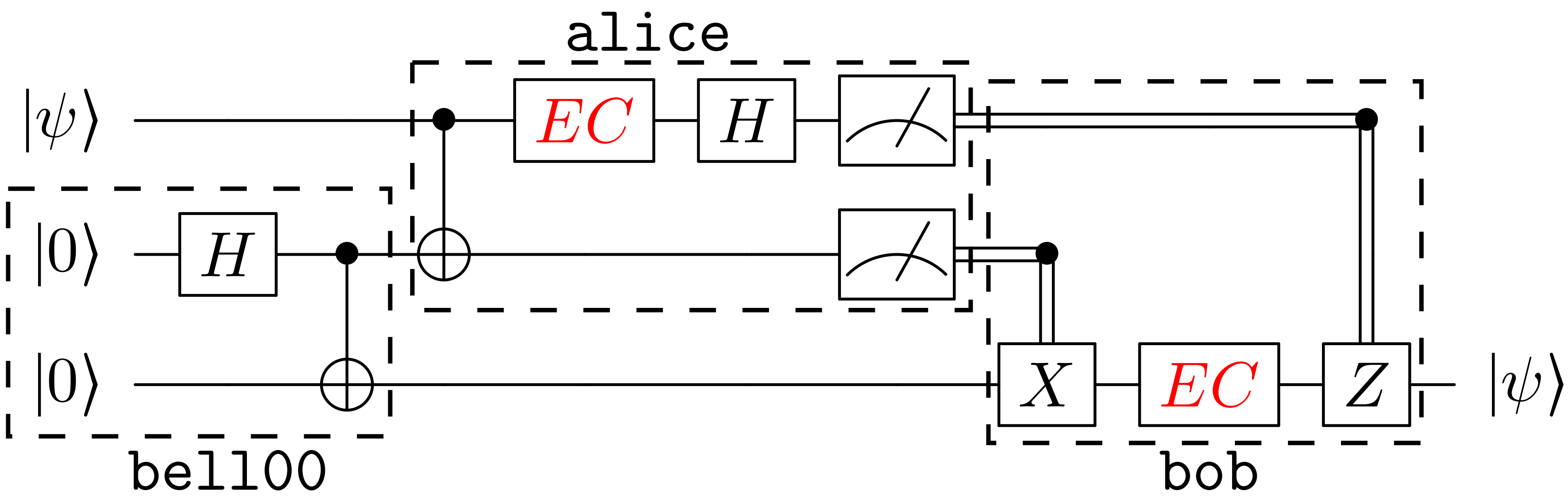

```
Definition bell : Box One (Qubit 2 ⊗ Qubit 2) := …

Definition alice : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit) := …
```

Typechecking



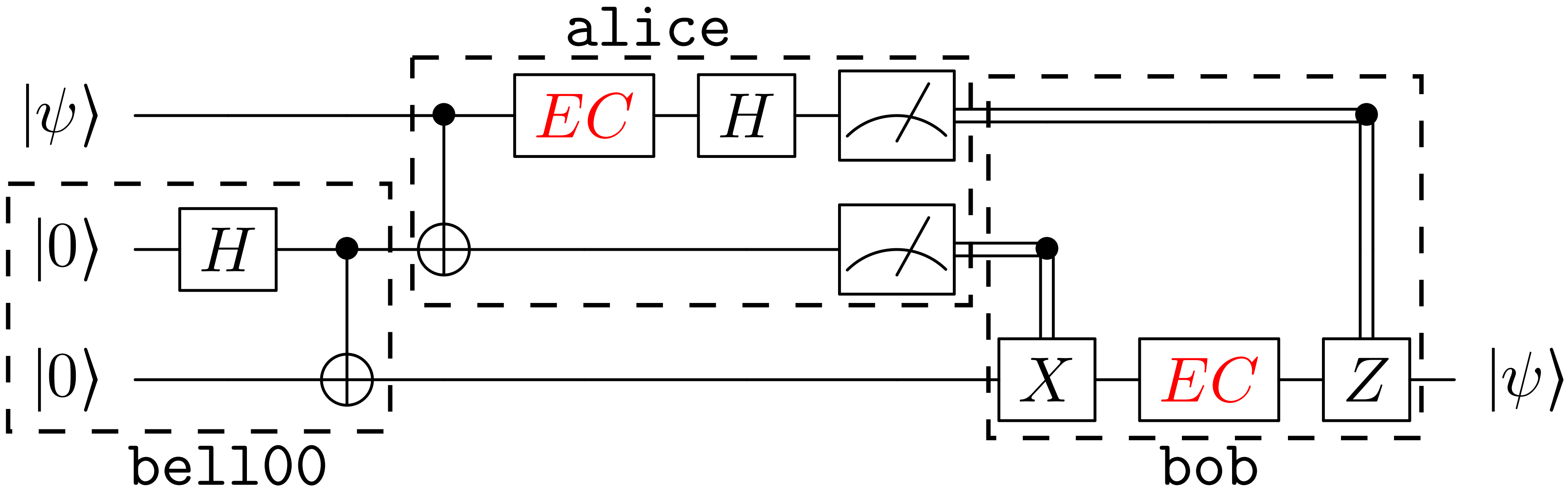

```
Definition bell : Box One (Qubit 2 ⊗ Qubit 2) := …

Definition alice : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit) := …

Definition bob : Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1) := …
```

Typechecking



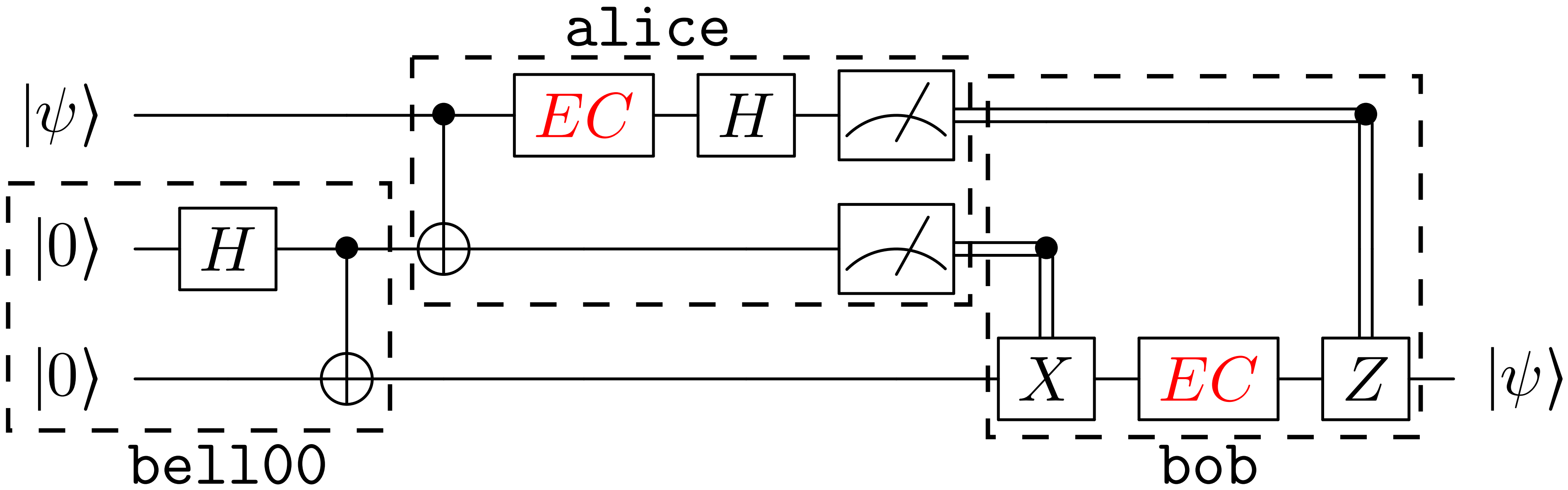

```
Definition bell : Box One (Qubit 2 ⊗ Qubit 2) := …

Definition alice : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit) := …

Definition bob : Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1) := …

Definition teleport : Box (Qubit 0) (Qubit 1) :=
```

Typechecking

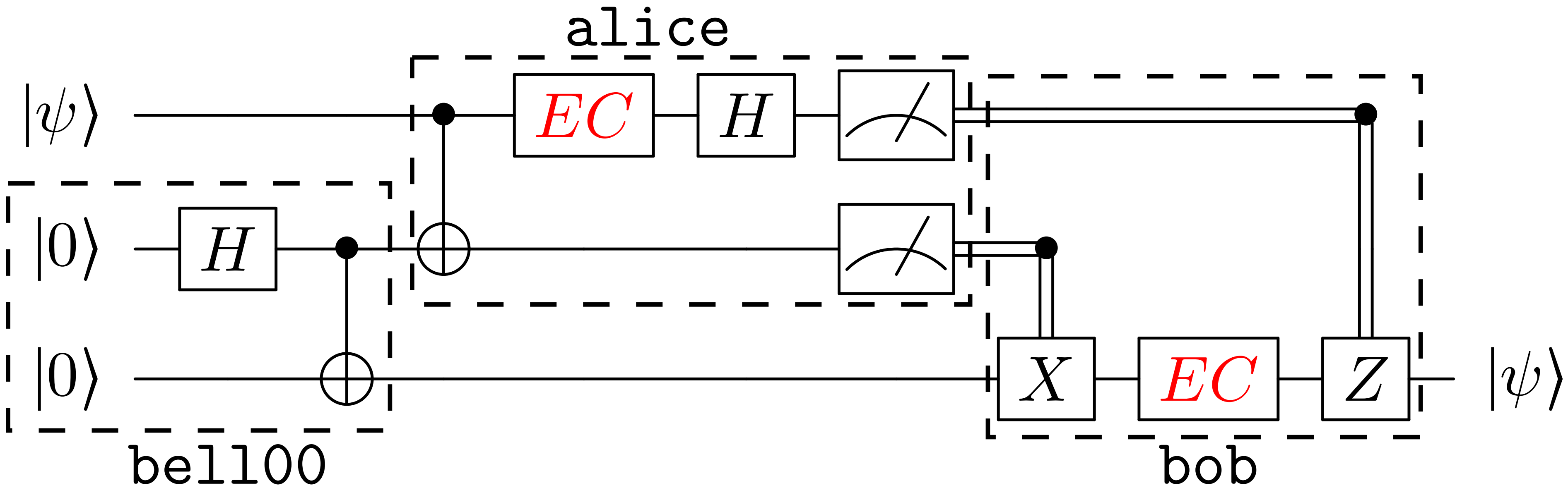



```
Definition bell : Box One (Qubit 2 ⊗ Qubit 2) := …

Definition alice : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit) := …

Definition bob : Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1) := …

Definition teleport : Box (Qubit 0) (Qubit 1) :=

Lemma teleport_EC_WT : Typed_Box teleport 3.
Proof. type_check. Qed.
```

Type Inference

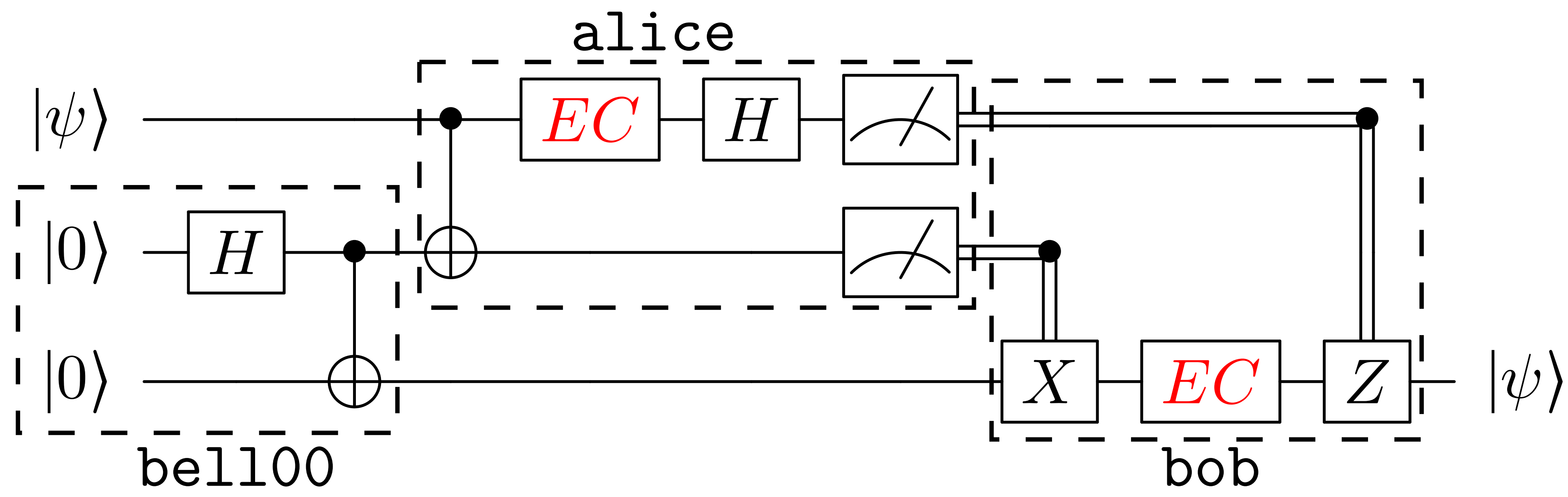


Type Inference




```
Definition bell := …
```

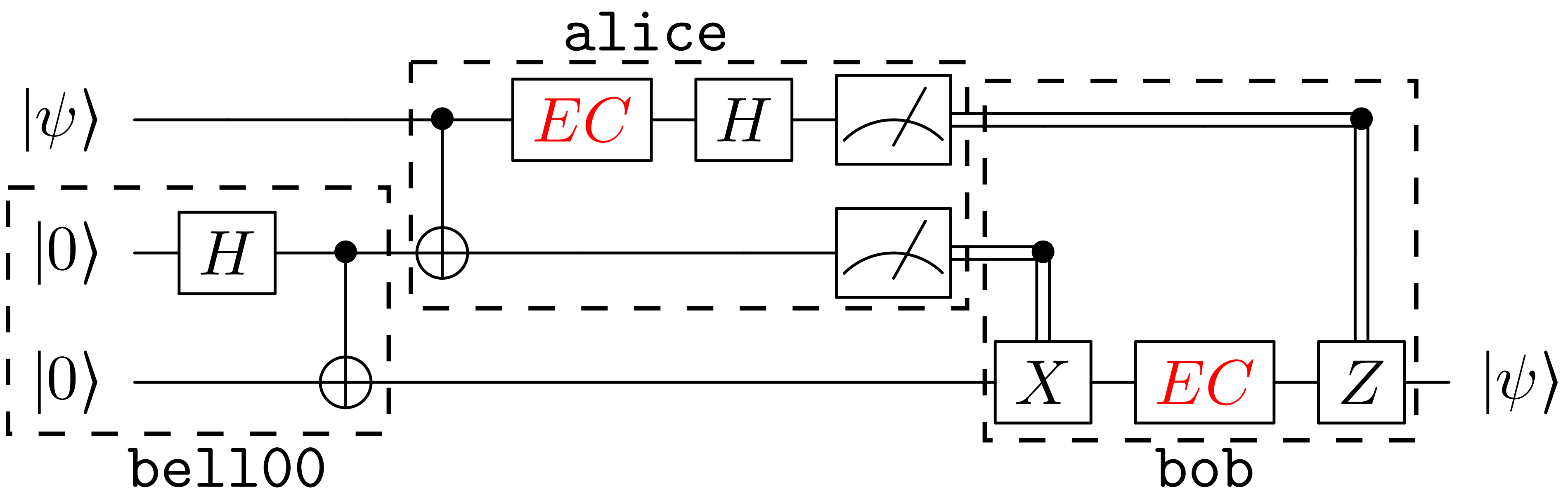
Type Inference



Check

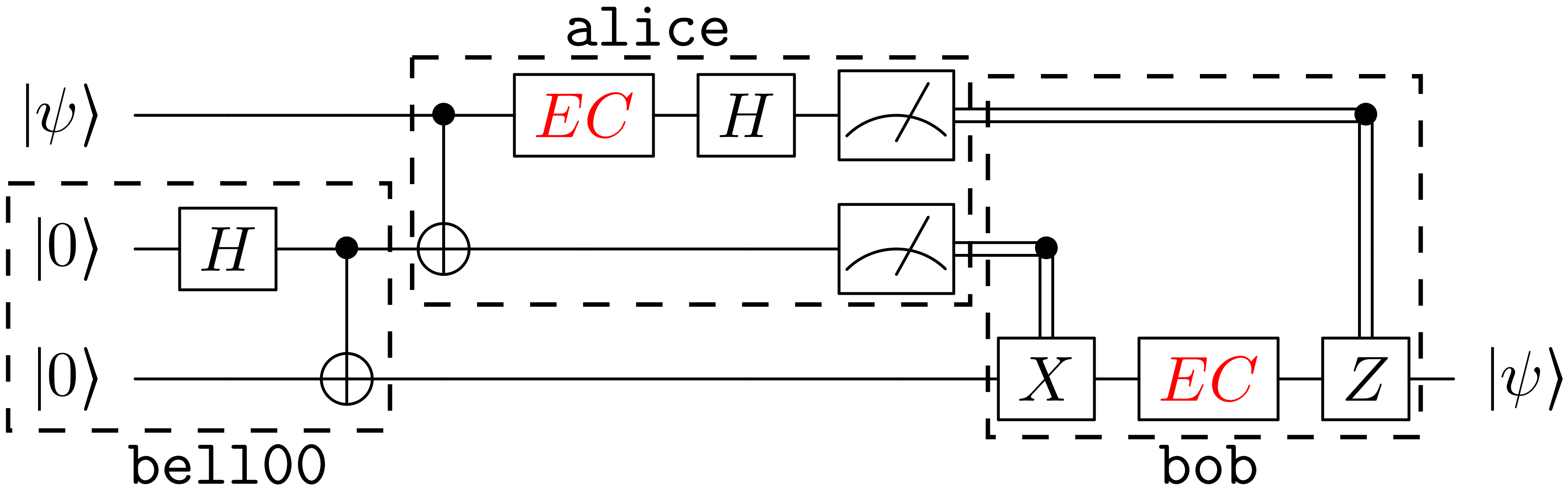
```
Definition bell := …
```

Type Inference




```
Check
Definition bell := …                          : Box One (Qubit 2 ⊗ Qubit 2)
```

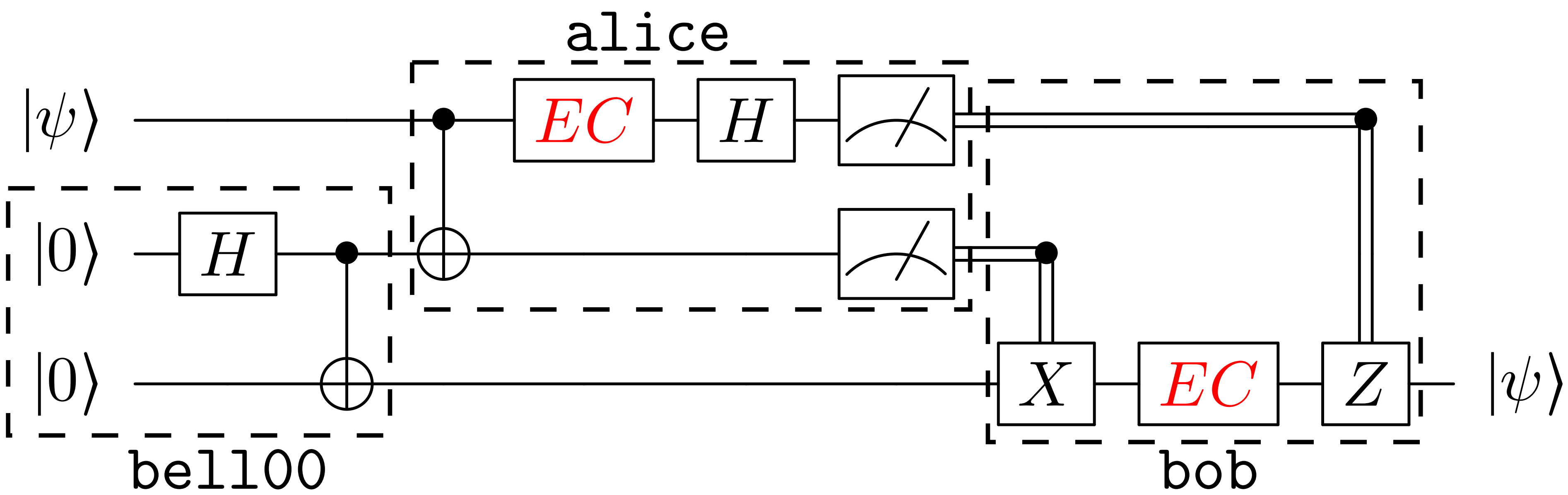
Type Inference



Check

```
Definition bell := …                       : Box One (Qubit 2 ⊗ Qubit 2)

Definition alice : _ (_ 0 ⊗ _ 2) _ := …
```

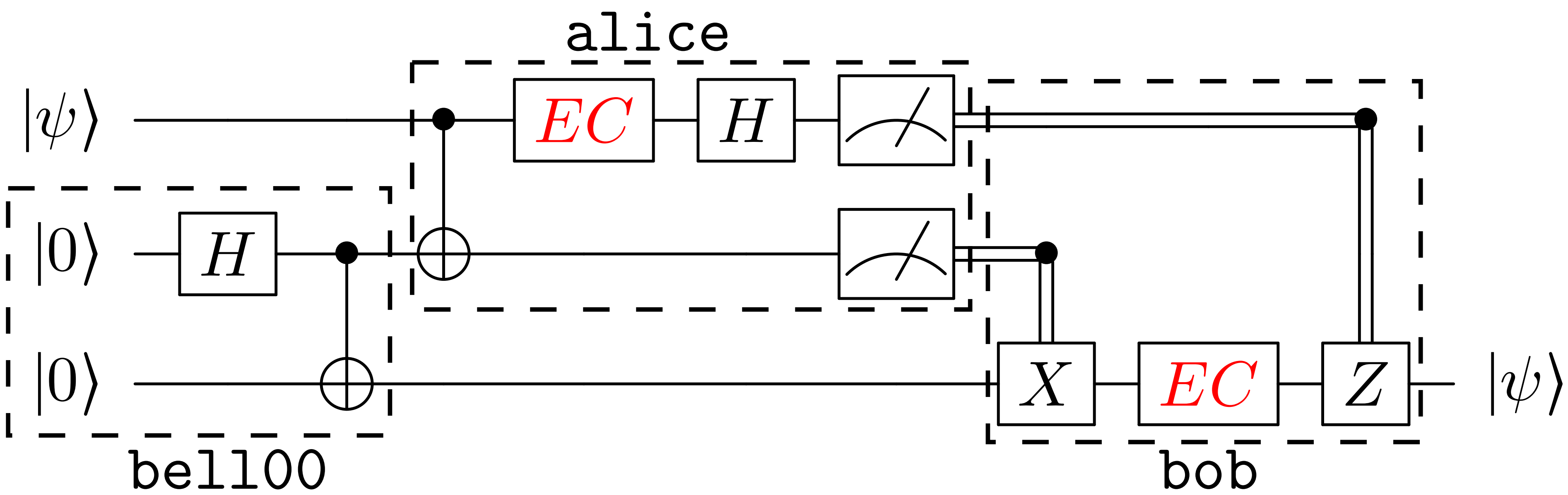
Type Inference



Check

```
Definition bell := …                              : Box One (Qubit 2 ⊗ Qubit 2)

Definition alice : _ (_ 0 ⊗ _ 2) _ := …           : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit)
```

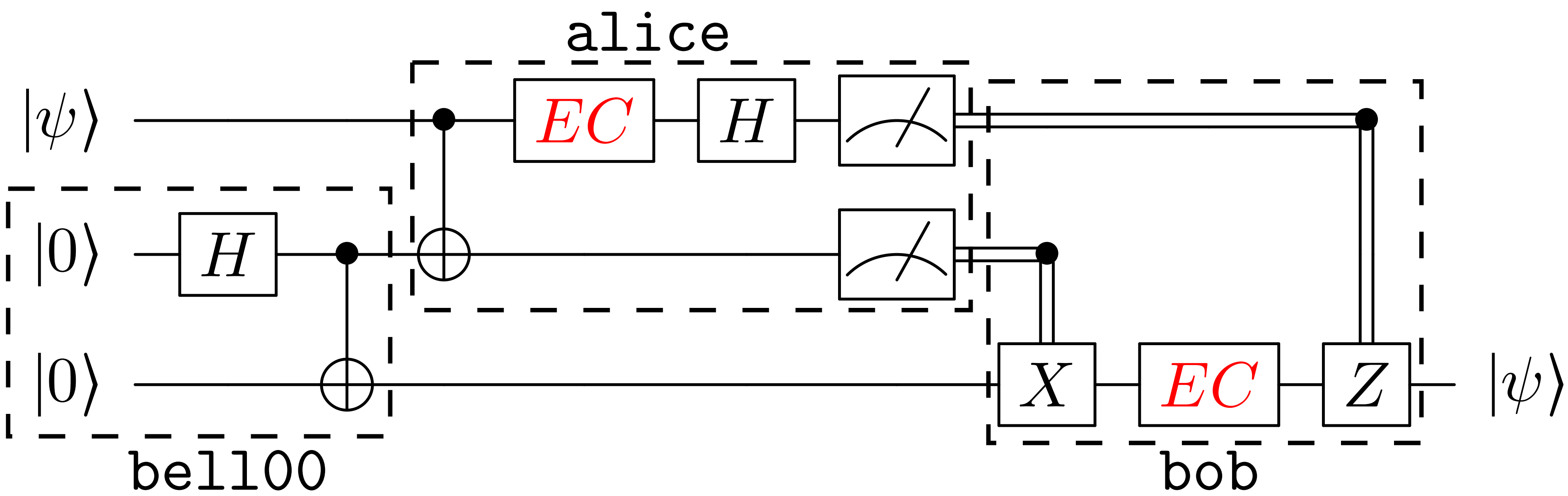
Type Inference



Check

```
Definition bell := …                                  : Box One (Qubit 2 ⊗ Qubit 2)

Definition alice : _ (_ 0 ⊗ _ 2) _ := …               : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit)

Definition bob : _ (_ ⊗ _ ⊗ _ 2) _ := …
```

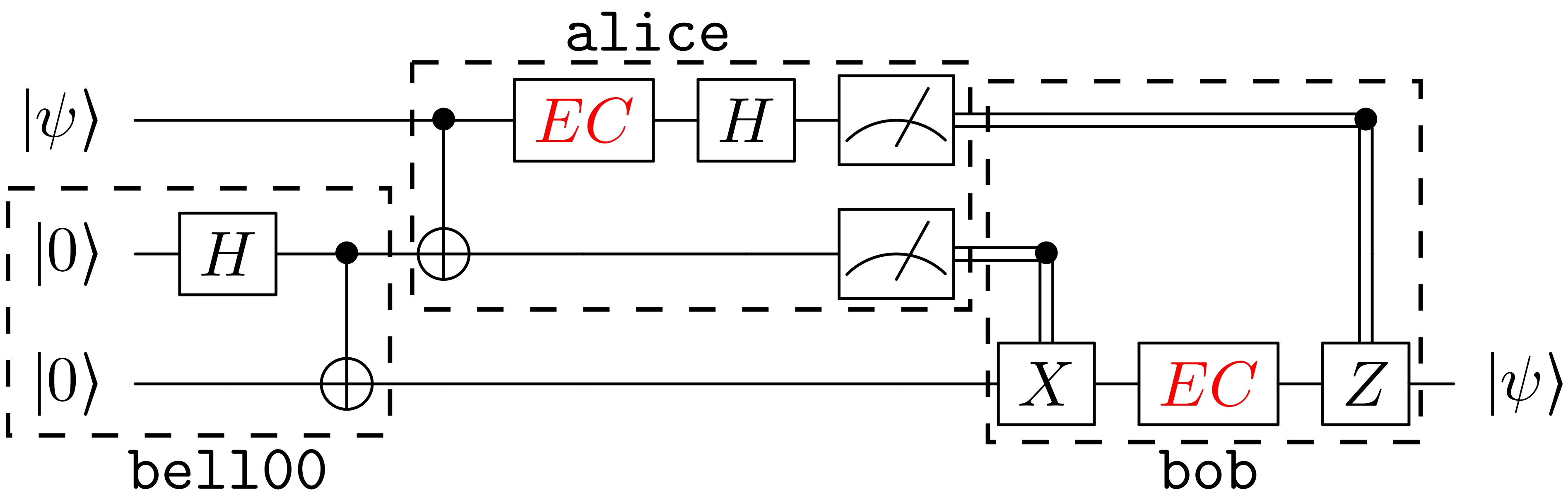
Type Inference



Check

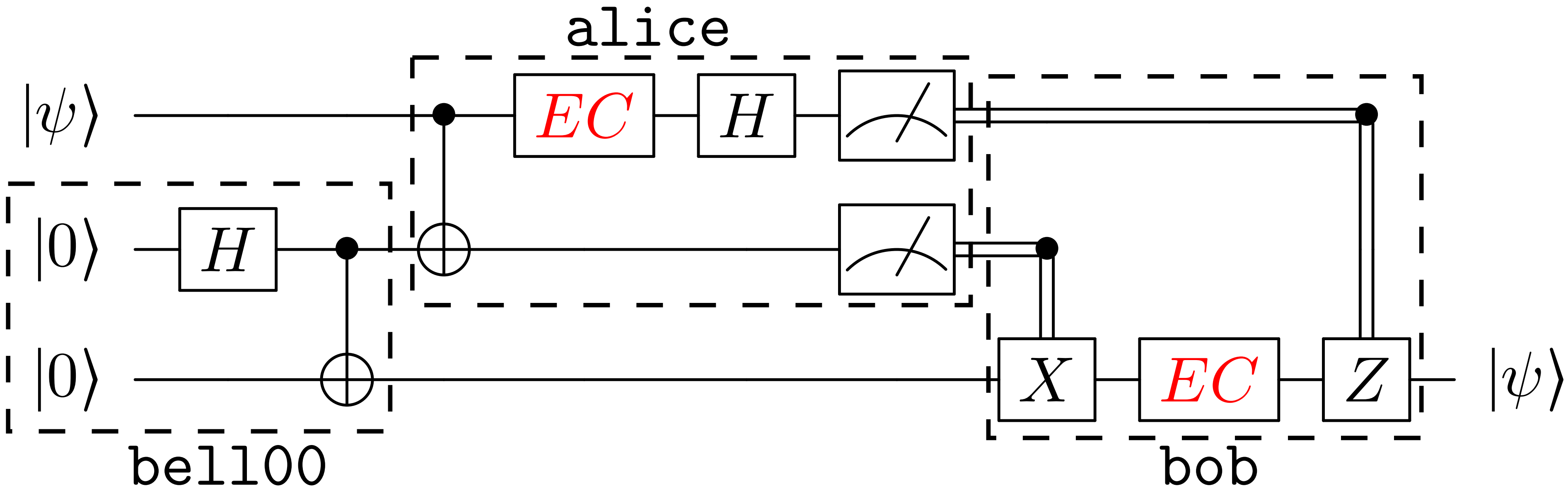
```
Definition bell := …                                  : Box One (Qubit 2 ⊗ Qubit 2)

Definition alice : _ (_ 0 ⊗ _ 2) _ := …               : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit)

Definition bob : _ (_ ⊗ _ ⊗ _ 2) _ := …               : Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1)
```

Type Inference




```
                                        Check
Definition bell := …                    : Box One (Qubit 2 ⊗ Qubit 2)

Definition alice : _ (_ 0 ⊗ _ 2) _ := … : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit)

Definition bob : _ (_ ⊗ _ ⊗ _ 2) _ := … : Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1)

Definition teleport := …
```

Type Inference



Check

```
Definition bell := …                                   : Box One (Qubit 2 ⊗ Qubit 2)

Definition alice : _ (_ 0 ⊗ _ 2) _ := …                : Box (Qubit 0 ⊗ Qubit 2) (Bit ⊗ Bit)

Definition bob : _ (_ ⊗ _ ⊗ _ 2) _ := …                : Box (Bit ⊗ Bit ⊗ Qubit 2) (Qubit 1)

Definition teleport := …                               : Box (Qubit 0) (Qubit 1)
```

Questions

- How can we reflect our error handling back into QWIRE's denotational semantics?
- What about errors due to decoherence, that aren't captured by gate application?
- What are better ways of handling errors in a type system or broader reasoning system?