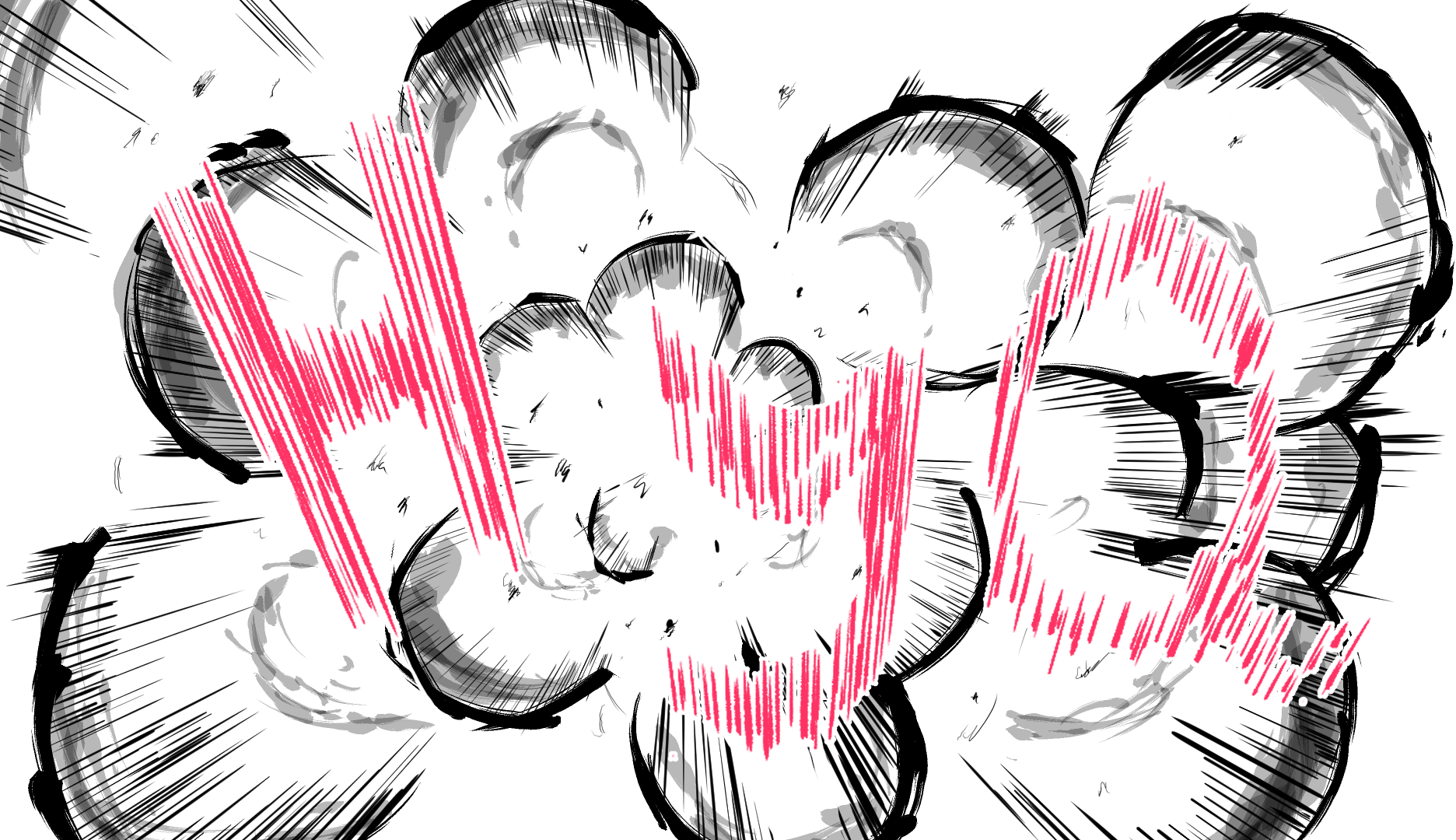


I have a pen

I have an apple...





Motivation

(for insiders)

HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

Motivation

(for insiders)

The first "interesting" instantiation
of Adjoint Logic

Motivation

(for insiders)

The first "interesting" instantiation
of Adjoint Logic

- 5 modes ← (only 3 in this talk)
- "realistic"

(basic) QRAM Model

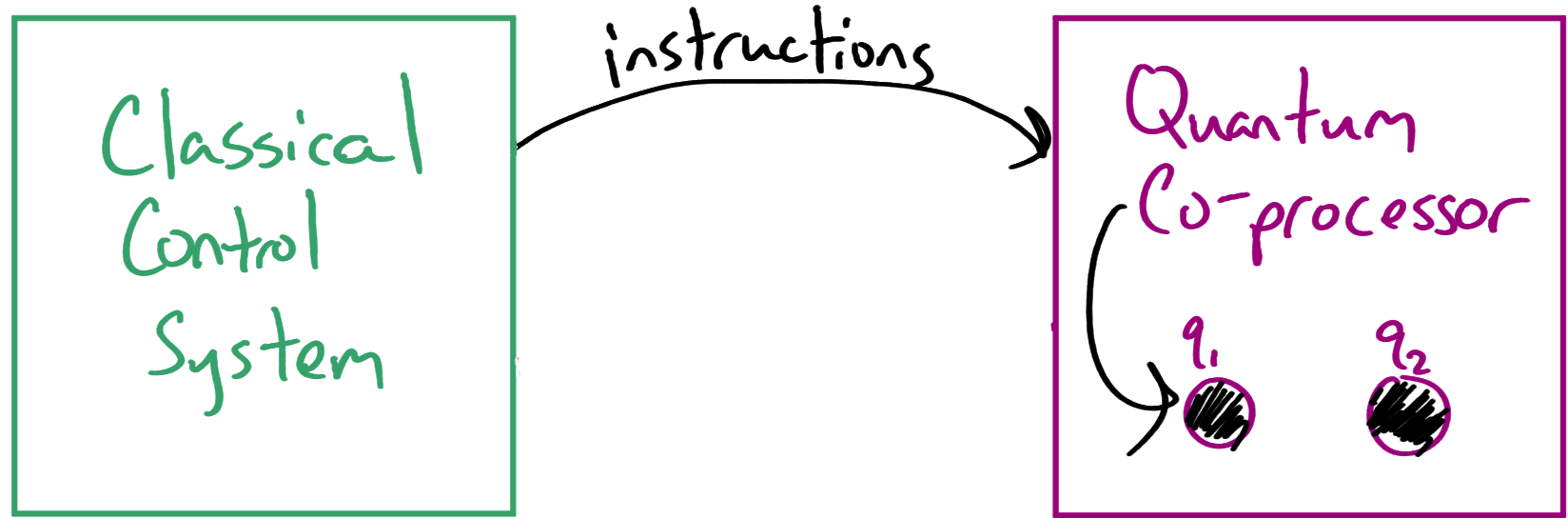
Classical
Control
System

Quantum
Co-processor

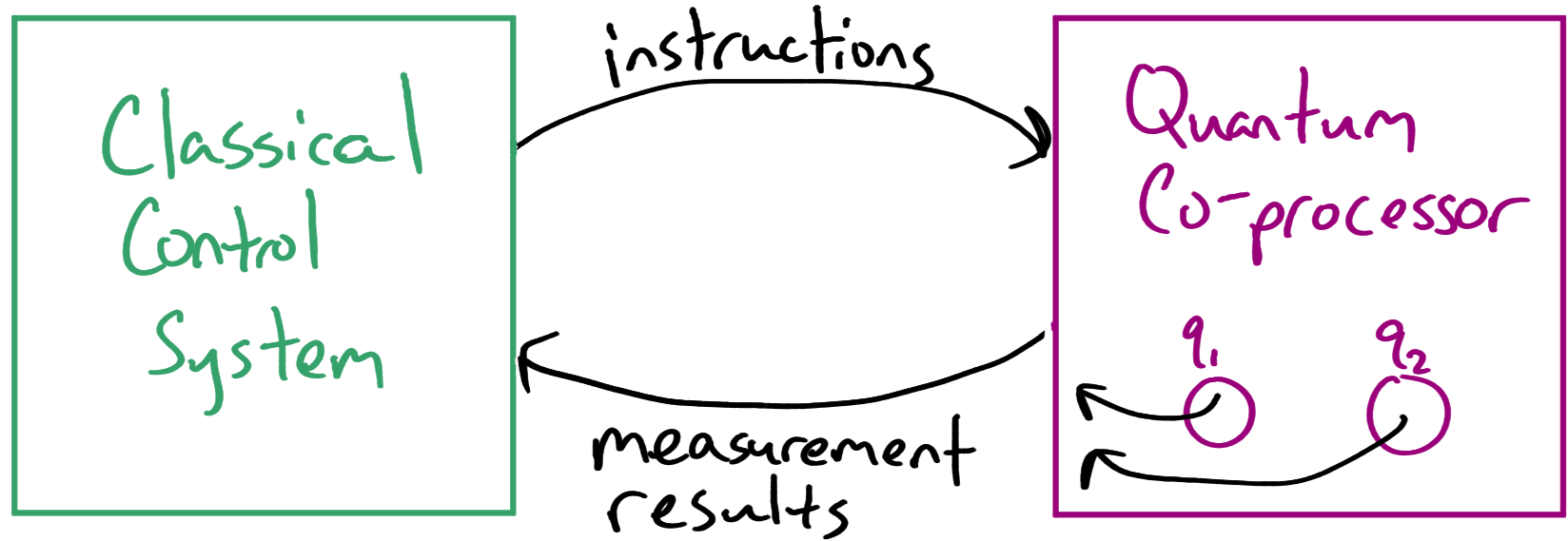
q_1
○

q_2
○

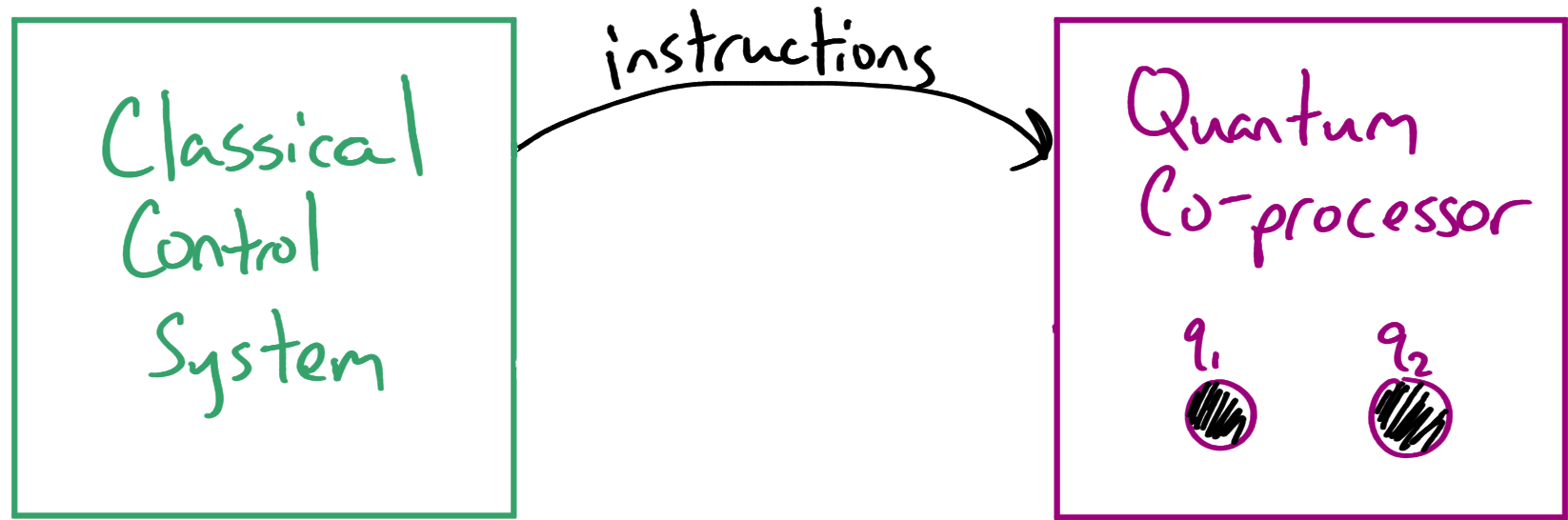
(basic) QRAM Model



(basic) QRAM Model



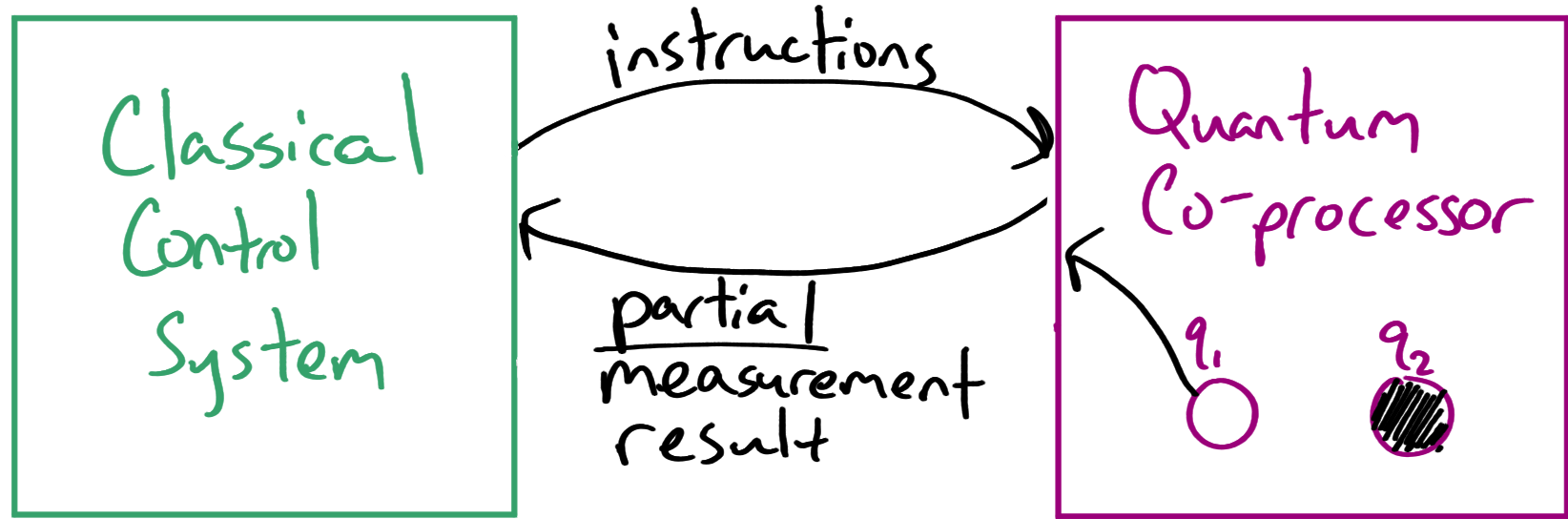
Hybrid Quantum-Classical Computing



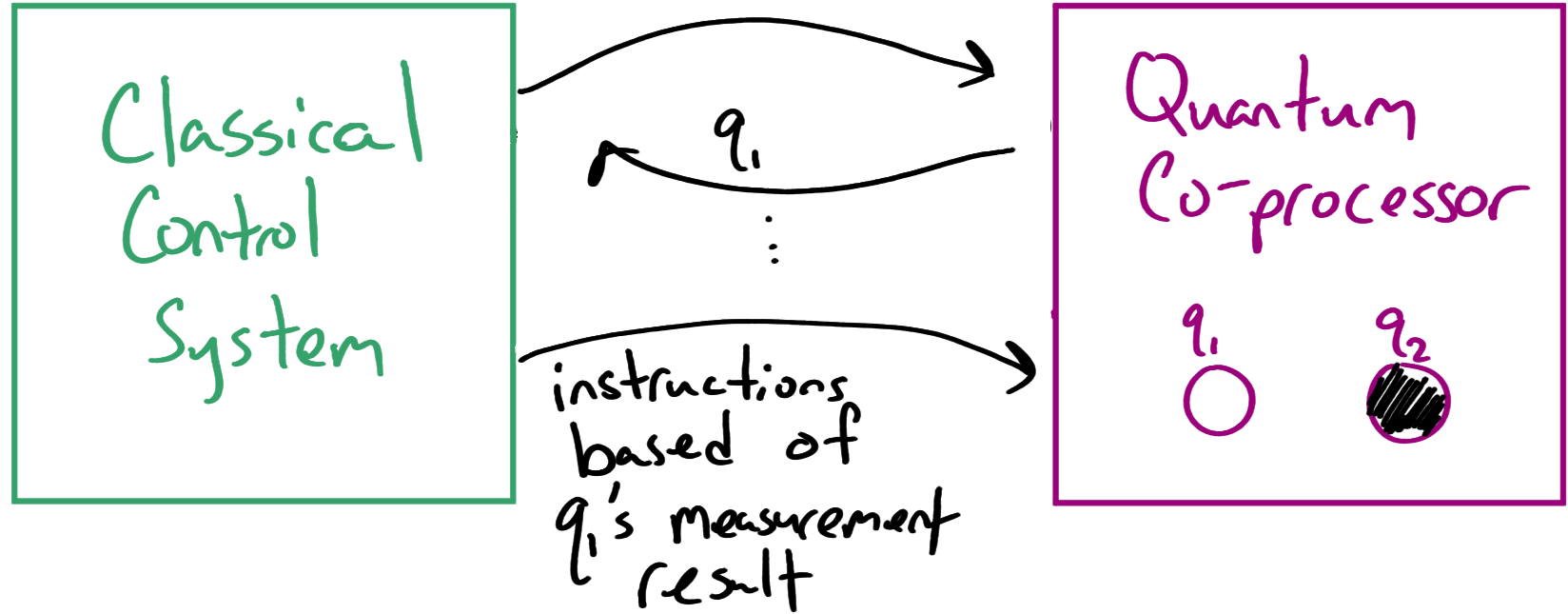
HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

Hybrid Quantum-Classical Computing



Hybrid Quantum-Classical Computing



HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

Hybrid Quantum-Classical Computing

Relevant for:

- Quantum Teleportation
- Error Correction
- Repeat-until-success

Hybrid Quantum-Classical Computing

Error Correction

HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

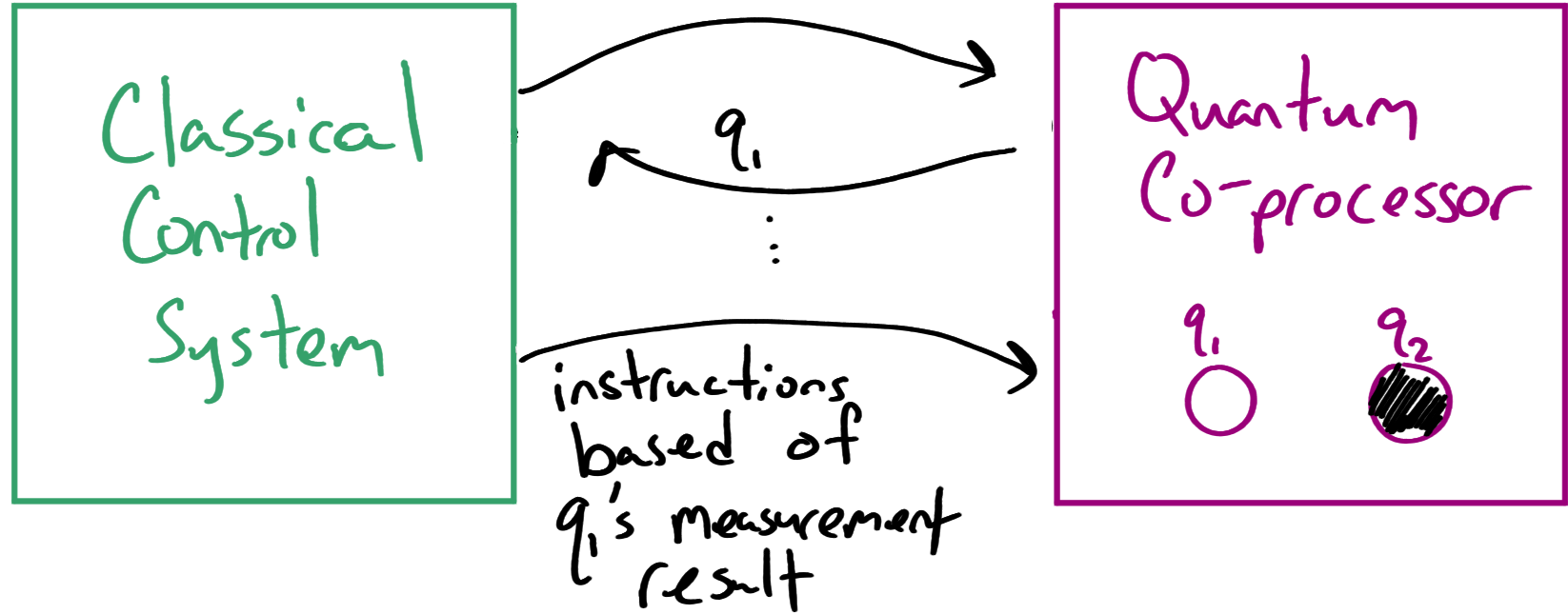
Hybrid Quantum-Classical Computing

Error Correction



Qubits are
very unstable
in practice!

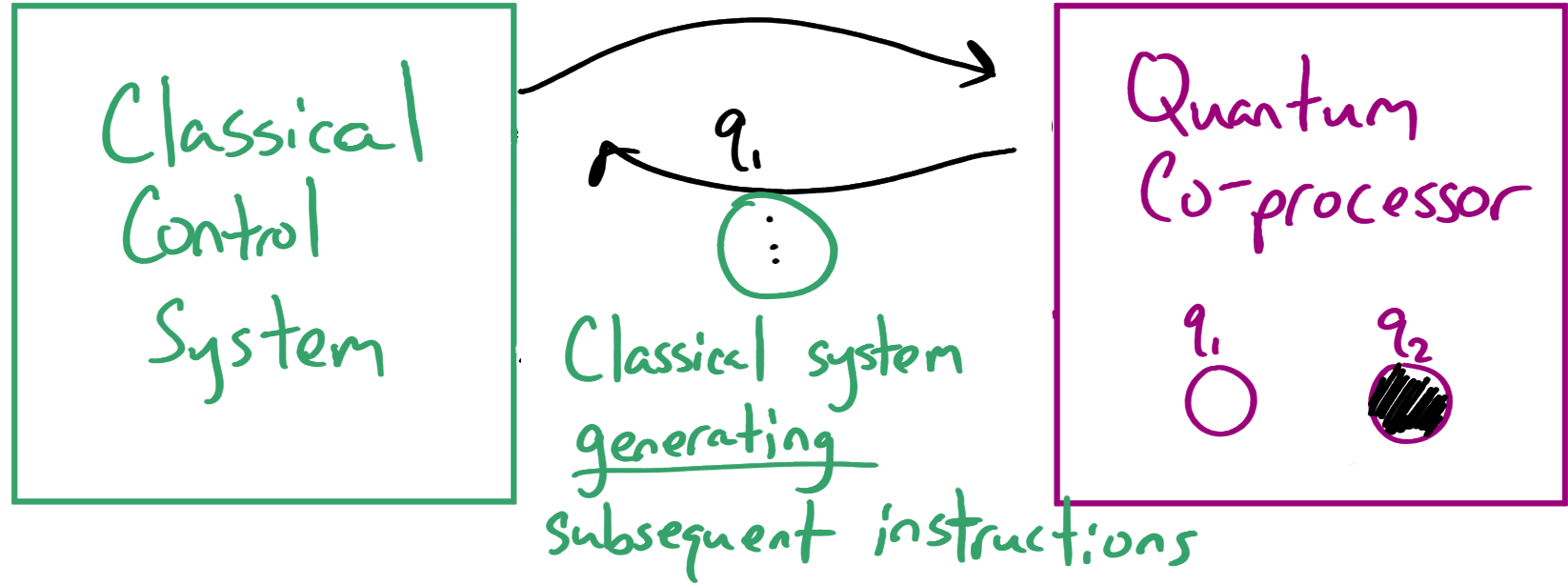
Hybrid Quantum-Classical Computing



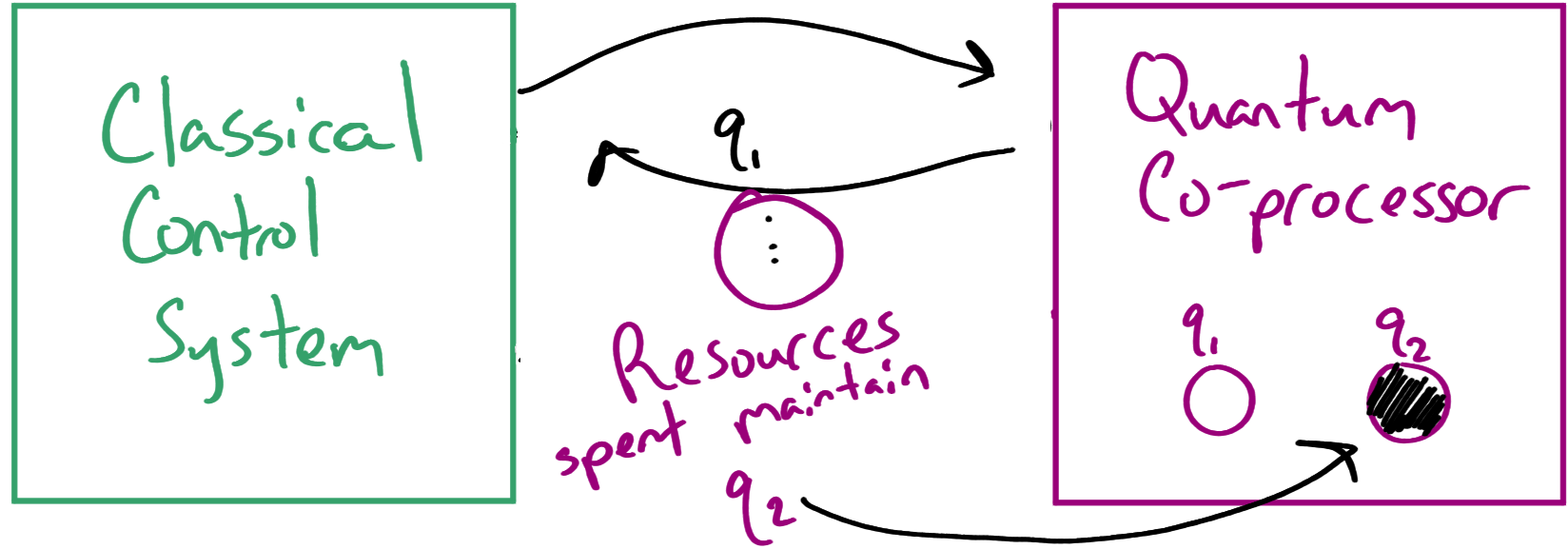
HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

Hybrid Quantum-Classical Computing



Hybrid Quantum-Classical Computing



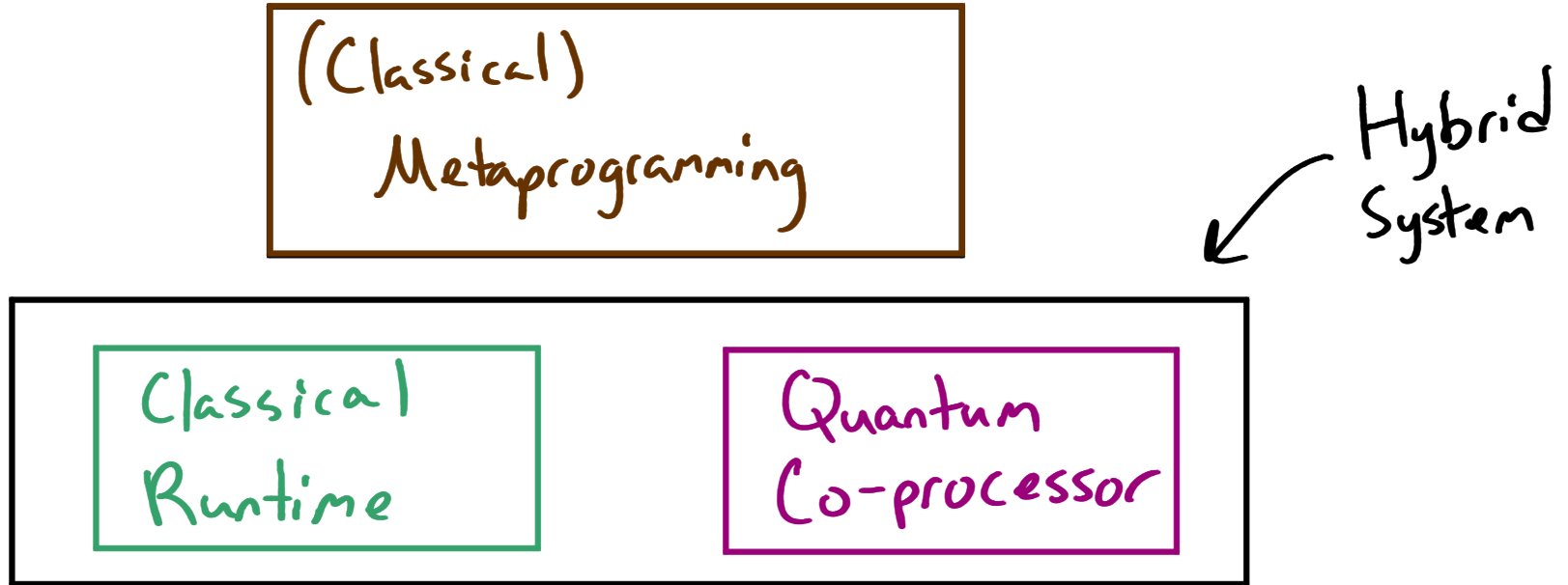
HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

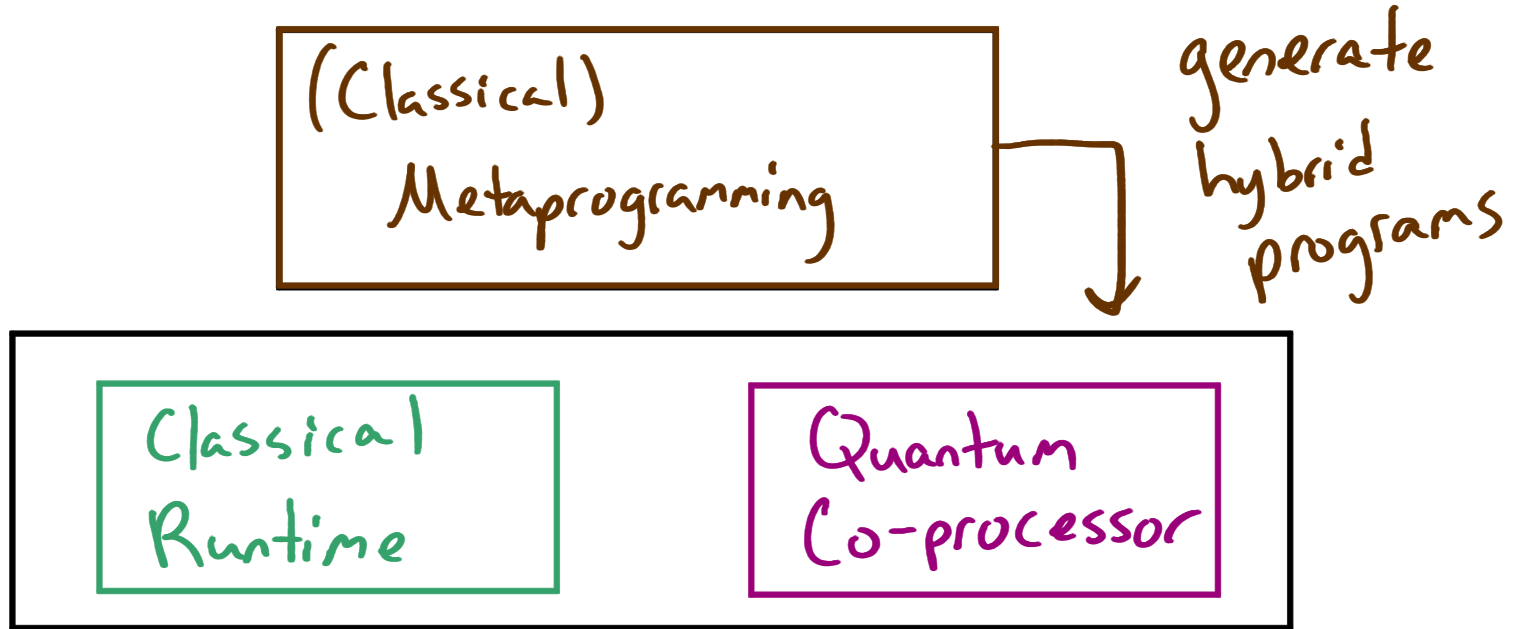
Motivation

Design a formal language that
ensures all **circuits** (instructions)
are generated before runtime

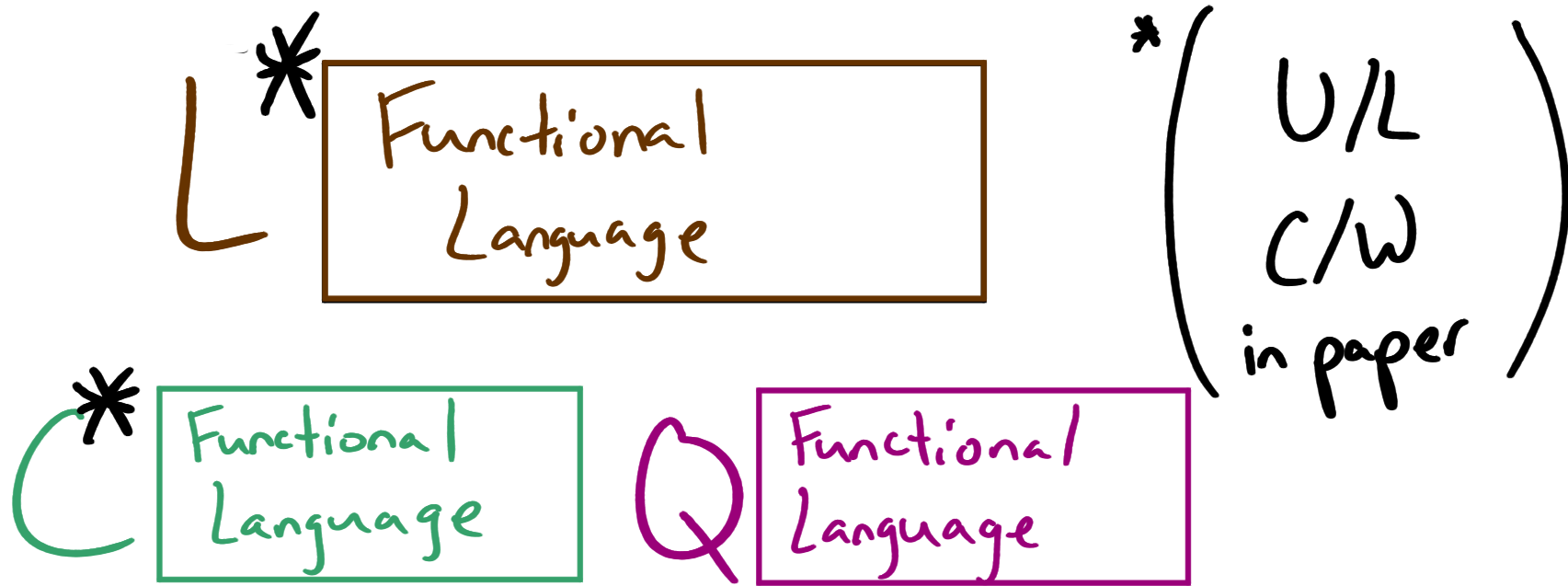
Intuition



Intuition



Technically:



Technically:

L Functional Language ✓

C Functional Language ✓

~~Q~~ Functional Language

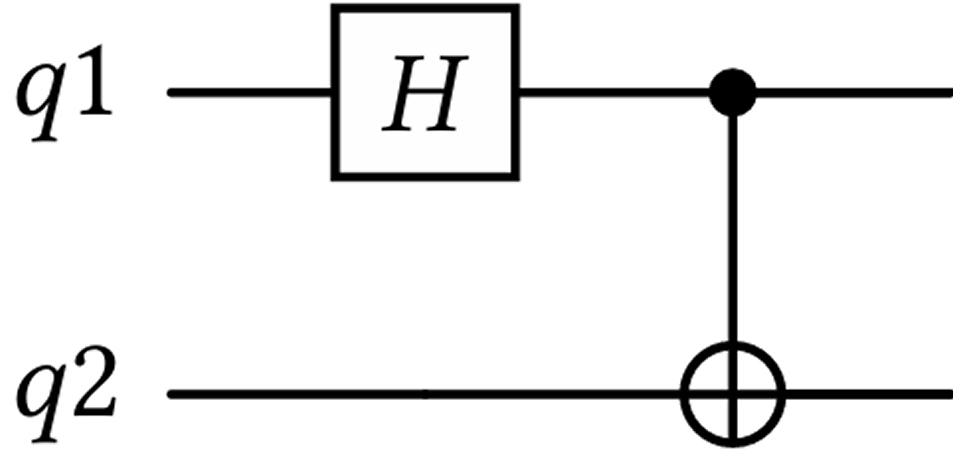
Technically:

L Functional Language ✓

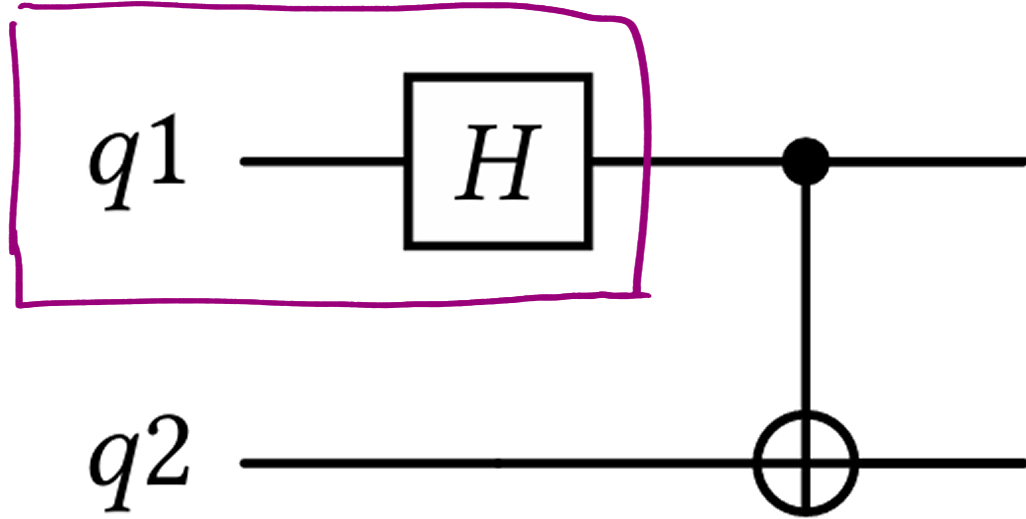
C Functional Language ✓

Q Functional Language ?

Quantum Circuits as Linear Functions



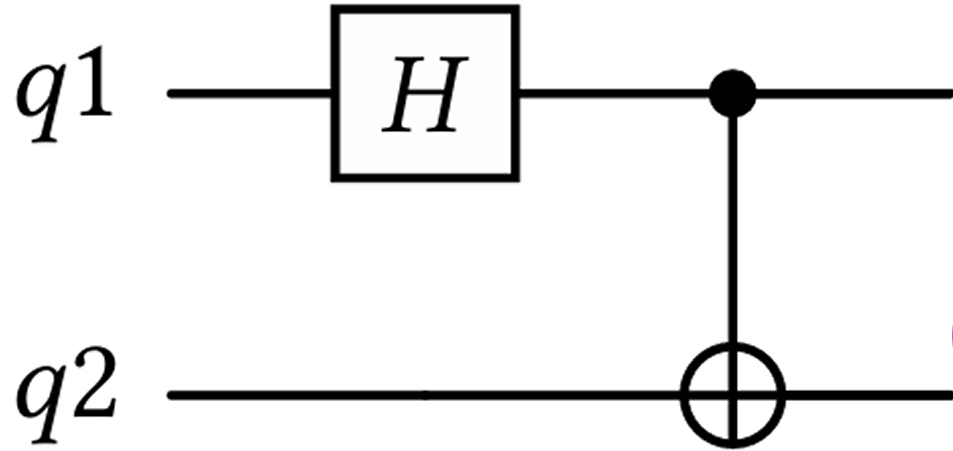
Quantum Circuits as Linear Functions



function application!

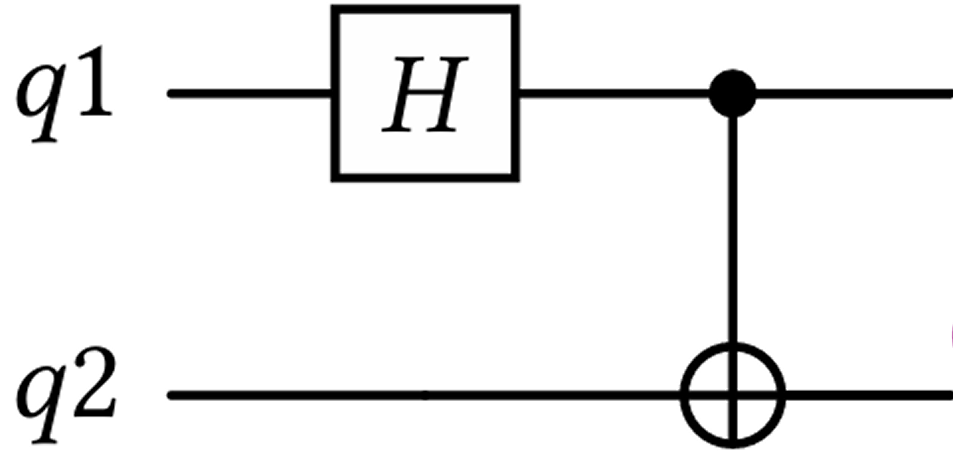
$H q_1$

Quantum Circuits as Linear Functions



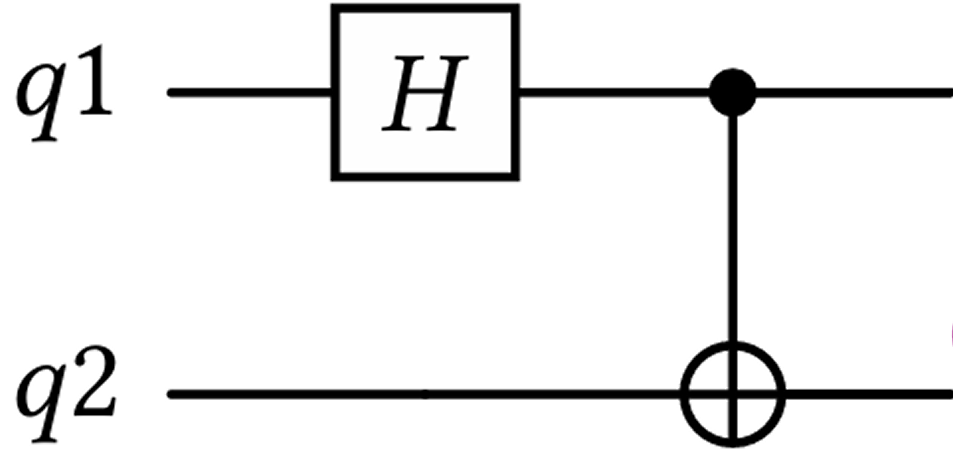
$CNOT \langle H q_1, q_2 \rangle$

Quantum Circuits as Linear Functions



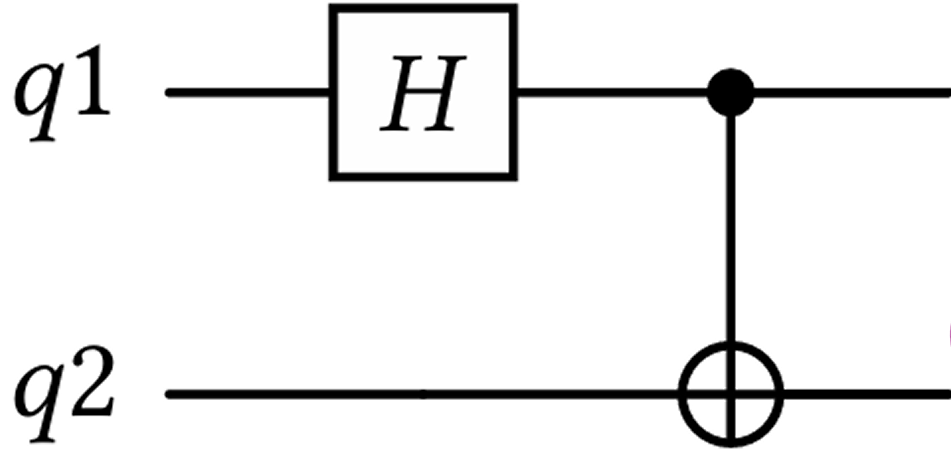
$\lambda x.$
let $\langle q_1, q_2 \rangle = x$ in
CNOT $\langle H q_1, q_2 \rangle$

Quantum Circuits as Linear Functions



$\lambda x.$ $\swarrow$ qubit $\otimes$ qubit
let $\langle q_1, q_2 \rangle = x$ in
CNOT $\langle H q_1, q_2 \rangle$

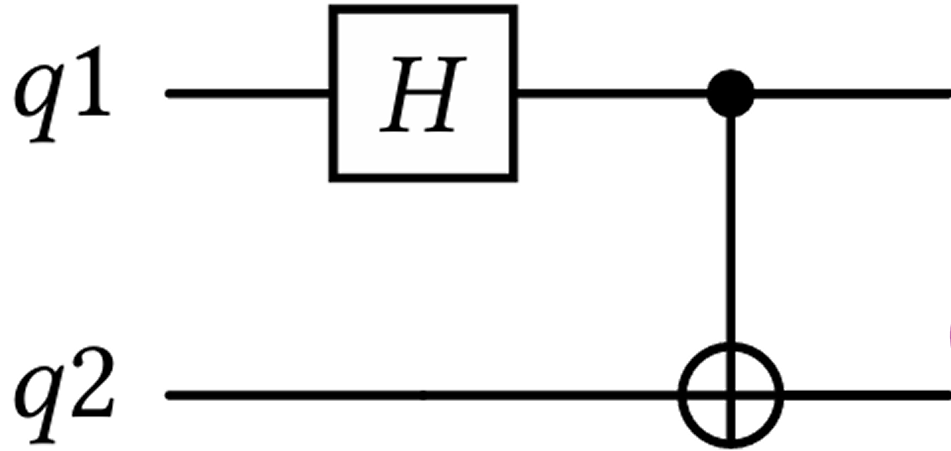
Quantum Circuits as Linear Functions



$\lambda x.$
let $\langle q_1, q_2 \rangle = x$ in
CNOT $\langle H q_1, q_2 \rangle$

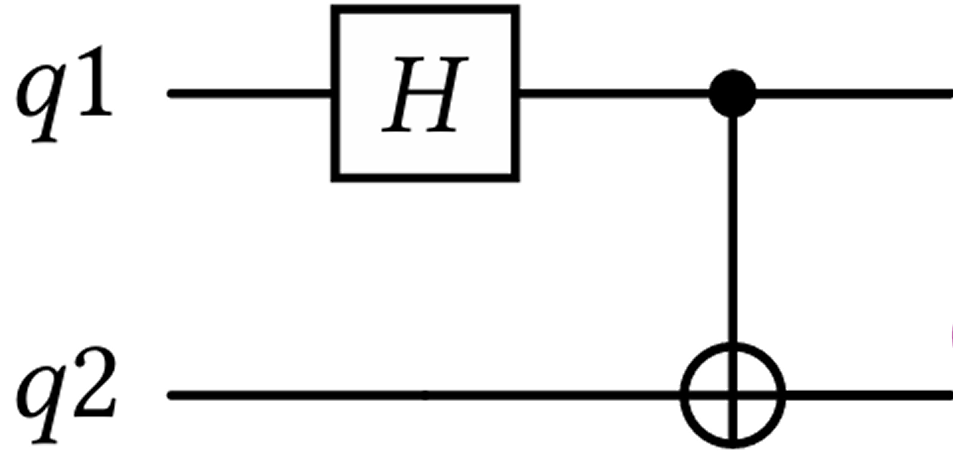
qubit

Quantum Circuits as Linear Functions



$\lambda x.$
let $\langle q_1, q_2 \rangle = x$ in
CNOT $\langle H q_1, q_2 \rangle$
qubit $\rightarrow$ qubit

Quantum Circuits as Linear Functions



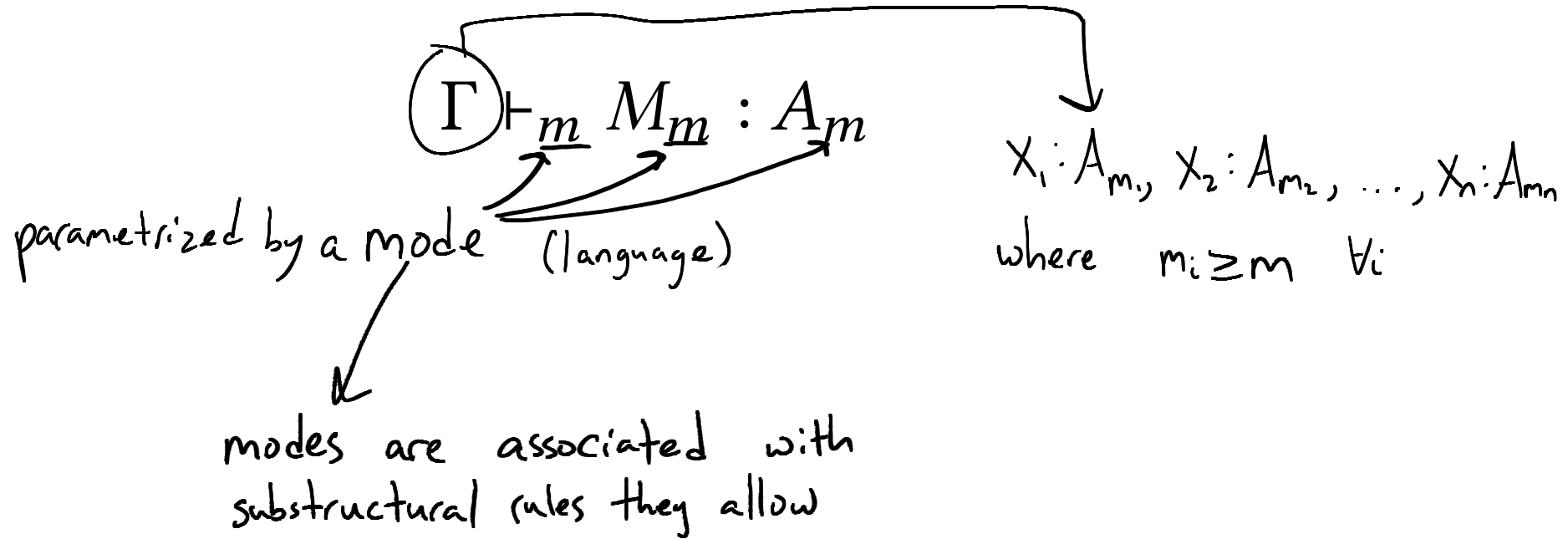
$\lambda x.$
let $\langle q_1, q_2 \rangle = x$ in
CNOT $\langle H q_1, q_2 \rangle$

$(\text{qubit} \otimes \text{qubit}) \rightarrow (\text{qubit} \otimes \text{qubit})$

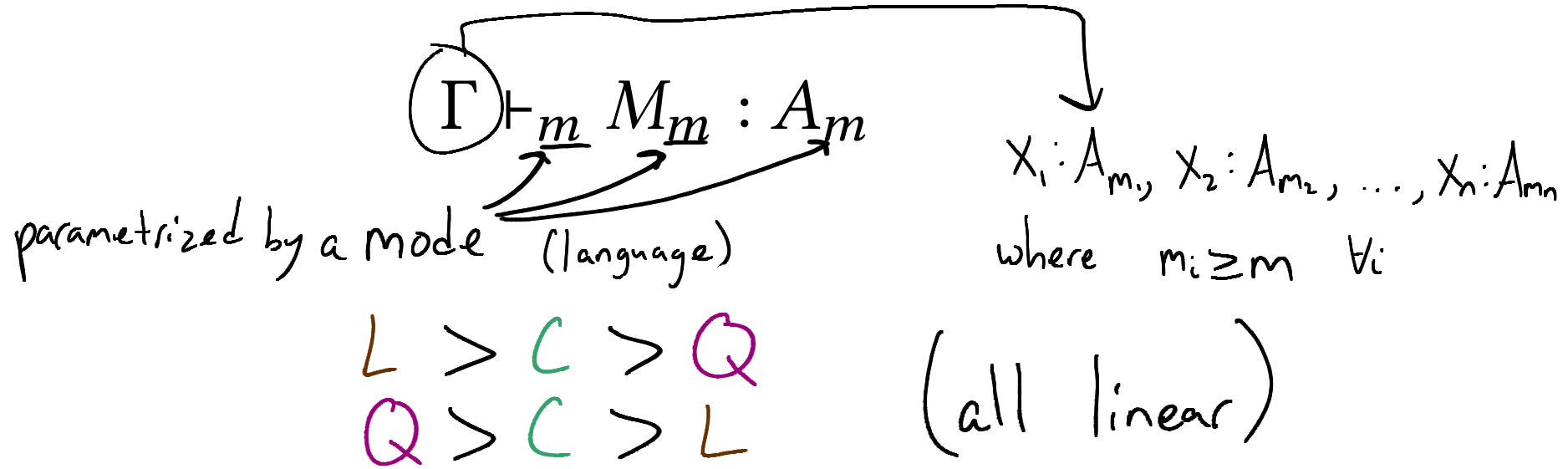
Adjoint Logic

$$\Gamma \vdash_m M_m : A_m$$

Adjoint Logic



Adjoint Logic



Adjoint Logic

$$\Gamma \vdash_m M_m : A_m$$

$$\frac{\Gamma, x : A_m \vdash_m M_m : B_m}{\Gamma \vdash_m \lambda x. M_m : A_m \multimap_m B_m} \text{ lam}$$

Adjoint Logic

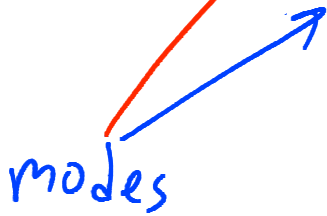
$$\Gamma \vdash_m M_m : A_m$$

$$\frac{\Gamma, x : A_m \vdash_m M_m : B_m}{\Gamma \vdash_m \lambda x. M_m : A_m \multimap_m B_m} \text{ lam}$$

Adjoint Logic - $\uparrow$ Shift

$$\frac{\Gamma \vdash_{\textcircled{m}} M_{\textcircled{m}} : A_{\textcircled{m}} \quad \Gamma \geq n}{\Gamma \vdash_{\textcircled{n}} \text{susp}_{\textcircled{m}}^{\textcircled{n}}(M_{\textcircled{m}}) : \uparrow_{\textcircled{m}}^{\textcircled{n}} A_{\textcircled{m}}} \uparrow I$$

modes



"Suspend m code
and capture it in n "

Adjoint Logic - $\uparrow$ Shift

$$\frac{\Gamma \vdash_{\hat{m}} M_{\hat{m}} : A_{\hat{m}} \quad \Gamma \geq n}{\Gamma \vdash_{\hat{n}} \text{susp}_{\hat{m}}^{\hat{n}}(M_{\hat{m}}) : \uparrow_{\hat{m}}^{\hat{n}} A_{\hat{m}}} \uparrow I \qquad
 \frac{\Gamma \vdash_{\hat{n}} M_{\hat{n}} : \uparrow_{\hat{m}}^{\hat{n}} A_{\hat{m}} \quad \Gamma \geq n}{\Gamma \vdash_{\hat{m}} \text{force}_{\hat{m}}^{\hat{n}}(M_{\hat{n}}) : A_{\hat{m}}} \uparrow E$$

"force the execution of suspended code"

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

$$(qubit \otimes_Q qubit) \rightarrow_Q (qubit \otimes_Q qubit)$$

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

$\uparrow_Q \left((\text{qubit} \otimes_Q \text{qubit}) \rightarrow_Q (\text{qubit} \otimes_Q \text{qubit}) \right)$

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

```
parallel_bell = susp(fn x. let <q1, q2, q3, q4> = x in  
  <force(bell_circ) <q1, q2>,    % call bell_circ on q1, q2  
  force(bell_circ) <q3, q4> >) % call bell_circ on q3, q4
```

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

x: qubit $\otimes_Q \dots \otimes_Q$ qubit

```
parallel_bell = susp(fn x. let <q1, q2, q3, q4> = x in  
  <force(bell_circ) <q1, q2>, % call bell_circ on q1, q2  
  force(bell_circ) <q3, q4> >) % call bell_circ on q3, q4
```

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

```
parallel_bell = susp(fn x. let <q1, q2, q3, q4> = x in  
  <force(bell_circ) <q1, q2>,      % call bell_circ on q1, q2  
  force(bell_circ) <q3, q4> >) % call bell_circ on q3, q4  
  ↑  
  Q( .... )
```

Suspending Quantum Circuits

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)
```

```
parallel_bell = susp(fn x. let <q1, q2, q3, q4> = x in  
  <force(bell_circ) <q1, q2>, % call bell_circ on q1, q2  
  force(bell_circ) <q3, q4> > % call bell_circ on q3, q4
```

$\uparrow$
 $\text{to } (\dots)$
 $(\text{qubit} \otimes \text{qubit}) \rightarrow \text{qubit} \otimes \text{qubit}$

Adjoint Logic - $\downarrow$ Shift

$$\frac{\Gamma \vdash_{(n)} M_{(n)} : A_{(n)} \quad \Gamma \geq n}{\Gamma \vdash_{(m)} \text{down}_{(m)}^{(n)}(M_{(n)}) : \downarrow_{(m)}^{(n)} A_{(n)}} \downarrow I$$

"evaluate n term and capture it in m "

Adjoint Logic - $\downarrow$ Shift

$$\frac{\Gamma \vdash_{(n)} M_{(n)} : A_{(n)} \quad \Gamma \geq n}{\Gamma \vdash_{(m)} \text{down}_{(m)}^{(n)}(M_{(n)}) : \downarrow_{(m)}^{(n)} A_{(n)}} \downarrow I \quad \frac{\Gamma_1 \vdash_{(m)} M_{(m)} : \downarrow_{(m)}^{(n)} A_{(n)} \quad \Gamma_2, x : A_{(n)} \vdash_{(n)} N_{(n)} : B_{(n)} \quad \Gamma_1 \geq n}{\Gamma_1 \bowtie \Gamma_2 \vdash_{(n)} \text{letdown}_{(m)}^{(n)} x = M_{(m)} \text{ in } N_{(n)} : B_{(n)}} \downarrow E$$

"bind some n value to x "

Calling the Quantum Co-processor

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)  
bell_pair = susp((force bell_circ) <init <>, init <>>)
```

$\uparrow_Q$ (qubit $\otimes_Q$ qubit)

initialize qubit

Calling the Quantum Co-processor

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)  
bell_pair = susp((force bell_circ) <init <>, init <>>)
```

$\uparrow_Q$ (qubit $\otimes_Q$ qubit)

```
let downCQ<q1:qubit, q2:qubit> = downCQ(force bell_pair) in  
...
```

C program

Calling the Quantum Co-processor

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)  
bell_pair = susp((force bell_circ) <init <>, init <>>)
```

$\uparrow_Q$ (qubit $\otimes_Q$ qubit)

```
let downcQ<q1:qubit, q2:qubit> = downcQ(force bell_pair) in  
...
```

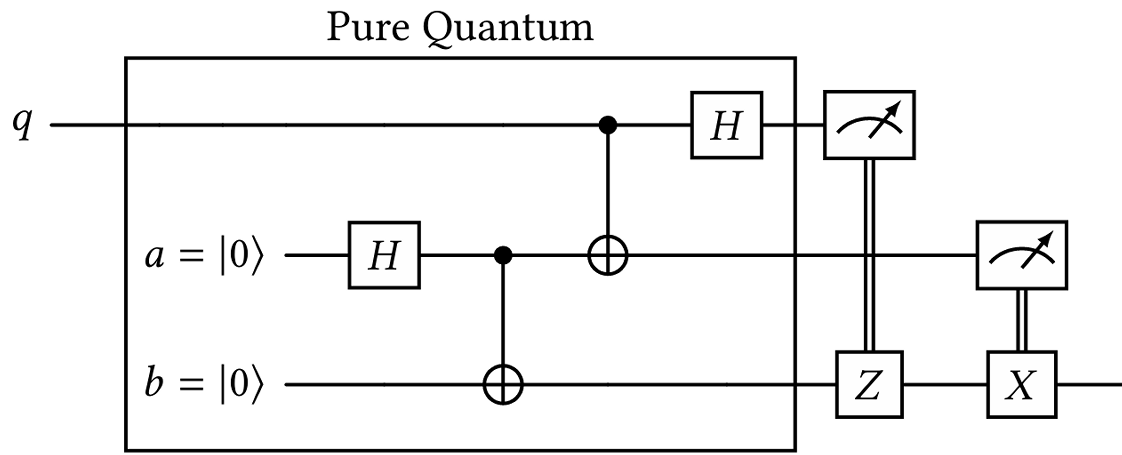
"request the quantum co-processor to
generate a bell pair"

Calling the Quantum Co-processor

```
bell_circ = susp(fn x. let <q1, q2> = x in CNOT <H q1, q2>)  
bell_pair = susp((force bell_circ) <init <>, init <>>)
```

```
let down <q1:qubit, q2:qubit> = down (force bell_pair) in  
if (measure q1) then % do something  
else % do something else
```

request quantum co-processor to
measure q_1 (hybrid)

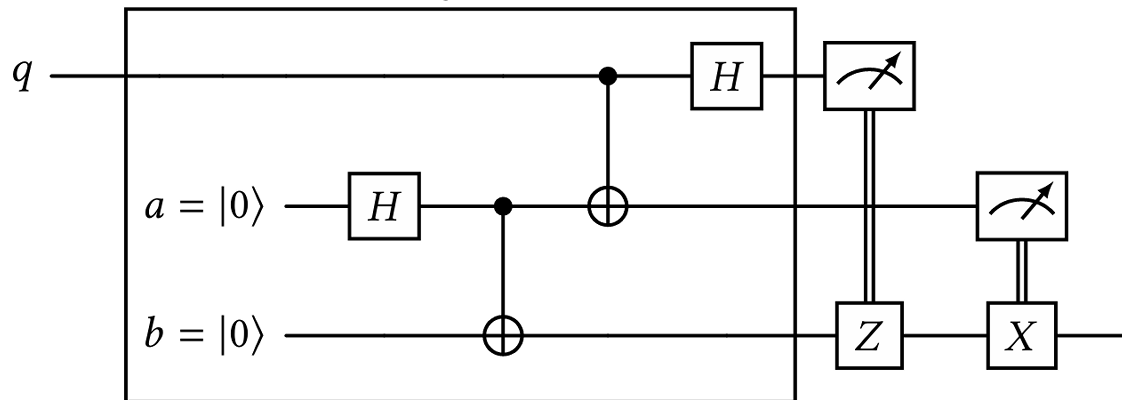


```

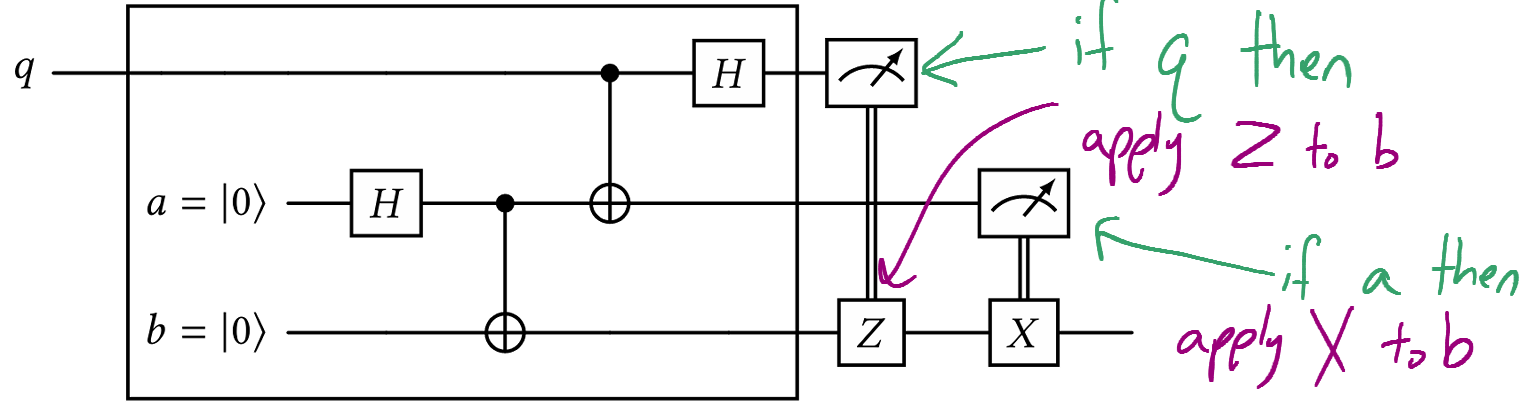
teleport = fn (q_ptr: down qubit).
  let down (q:qubit) = q_ptr in
  let down <a:qubit, b:qubit> = down (force bell_pair)
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
  let down (q:qubit) = down (H q) in
  let down (b:qubit) =
    if (measure q) then down (Z b) else down b
  in
  if (measure a) then down (X b) else down b

```

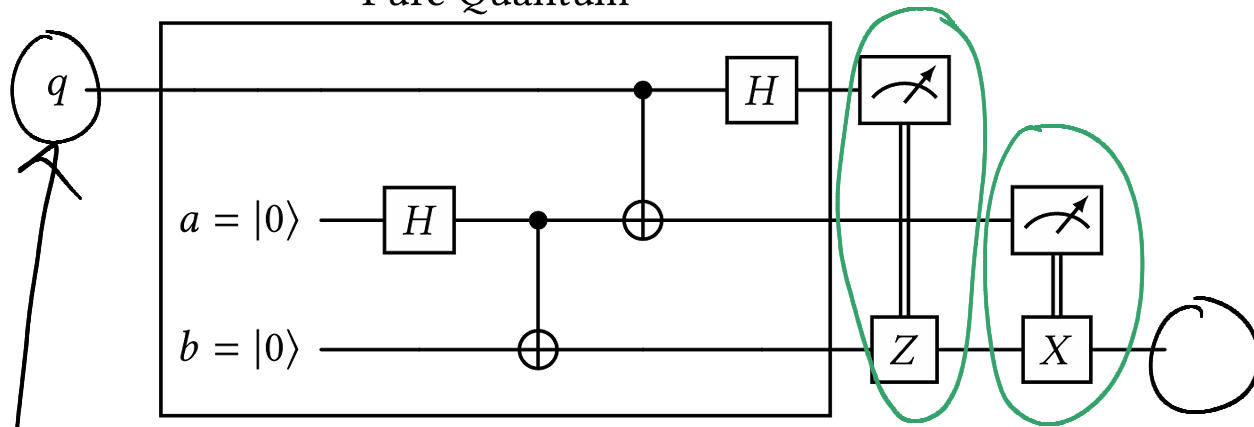
Pure Quantum



Pure Quantum



Pure Quantum



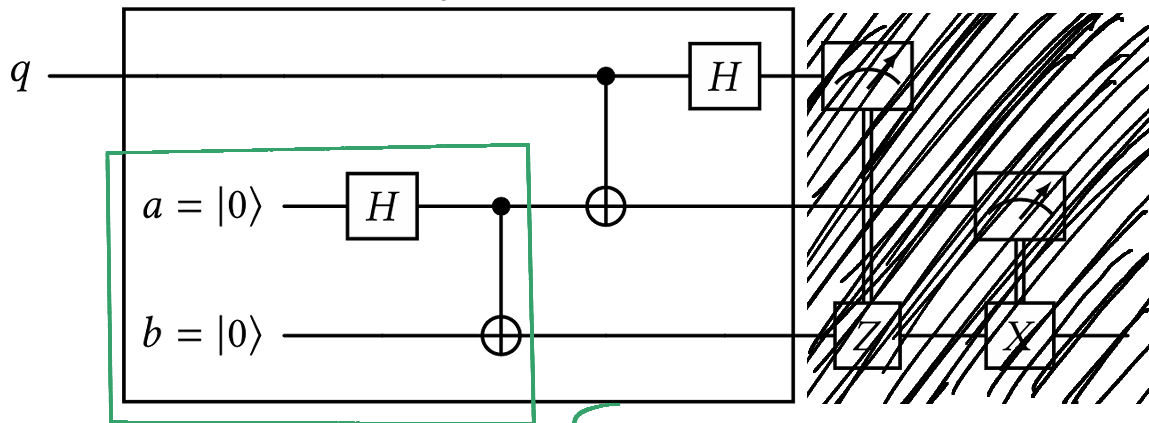
teleport = fn (q_ptr: down qubit).

Hybrid Program!

$\downarrow_c^Q$ qubit

$\rightarrow_c \downarrow_c^Q$ qubit

Pure Quantum



```
teleport = fn (q_ptr: down qubit).
```

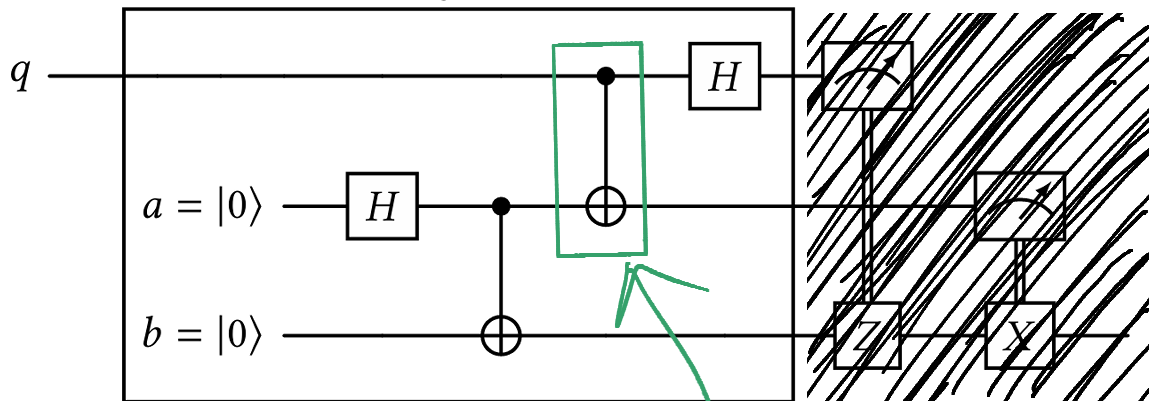
```
  let down (q:qubit) = q_ptr in
```

```
  let down <a:qubit, b:qubit> = down (force bell_pair)
```

```
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
```

```
  let down (q:qubit) = down (H q) in
```

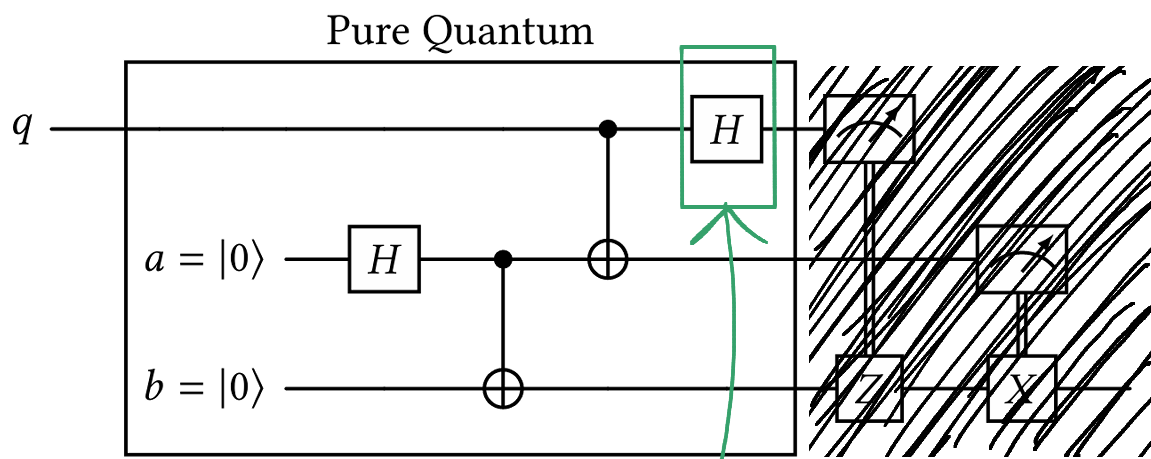
Pure Quantum



```

teleport = fn (q_ptr: down qubit).
  let down (q:qubit) = q_ptr in
  let down <a:qubit, b:qubit> = down (force bell_pair)
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
  let down (q:qubit) = down (H q) in

```

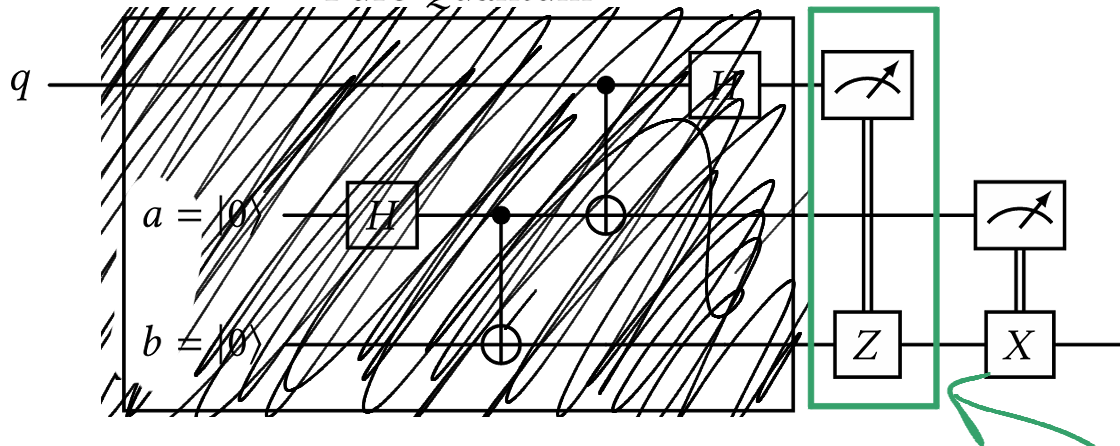


```

teleport = fn (q_ptr: down qubit).
  let down (q:qubit) = q_ptr in
  let down <a:qubit, b:qubit> = down (force bell_pair)
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
  let down (q:qubit) = down (H q) in

```

Pure Quantum



```
teleport = fn (q_ptr: down qubit).
```

```
  let down (q:qubit) = q_ptr in
```

```
  let down <a:qubit, b:qubit> = down (force bell pair)
```

```
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
```

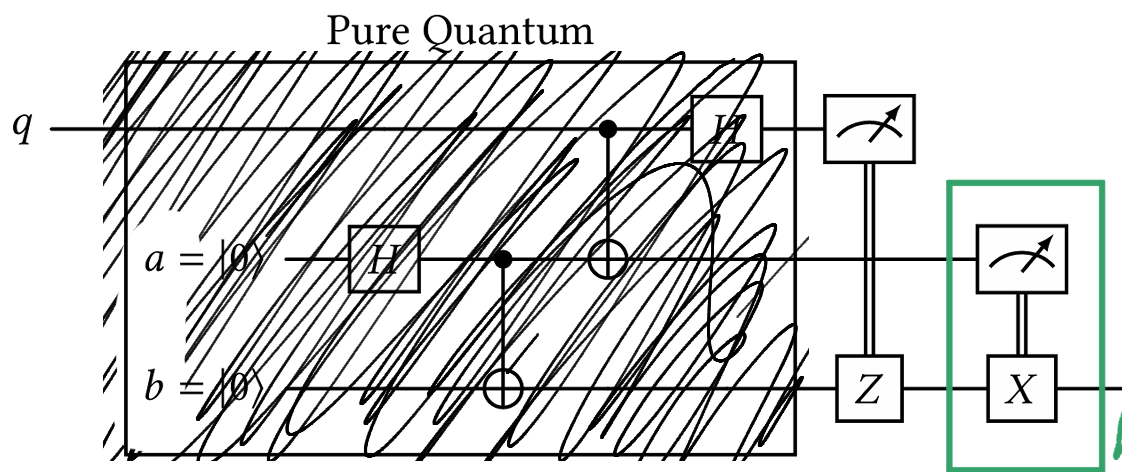
```
  let down (q:qubit) = down (H q) in
```

```
  let down (b:qubit) =
```

```
    if (measure q) then down (Z b) else down b
```

```
  in
```

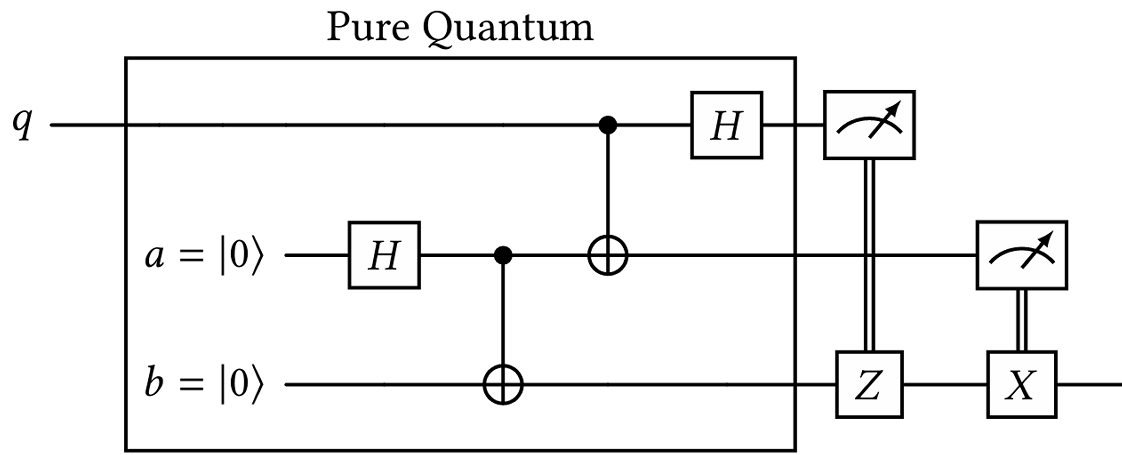
```
  if (measure a) then down (X b) else down b
```



```

teleport = fn (q_ptr: down qubit).
  let down (q:qubit) = q_ptr in
  let down <a:qubit, b:qubit> = down (force bell pair)
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
  let down (q:qubit) = down (H q) in
  let down (b:qubit) =
    if (measure q) then down (Z b) else down b
  in
  if (measure a) then down (X b) else down b

```

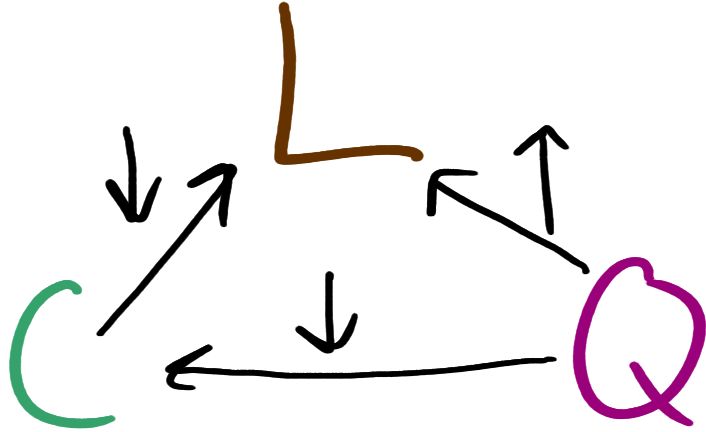


```

teleport = fn (q_ptr: down qubit).
  let down (q:qubit) = q_ptr in
  let down <a:qubit, b:qubit> = down (force bell_pair)
  let down <q:qubit, a:qubit> = down (CNOT <q, a>) in
  let down (q:qubit) = down (H q) in
  let down (b:qubit) =
    if (measure q) then down (Z b) else down b
  in
  if (measure a) then down (X b) else down b

```

Meta programming Hybrid Programs

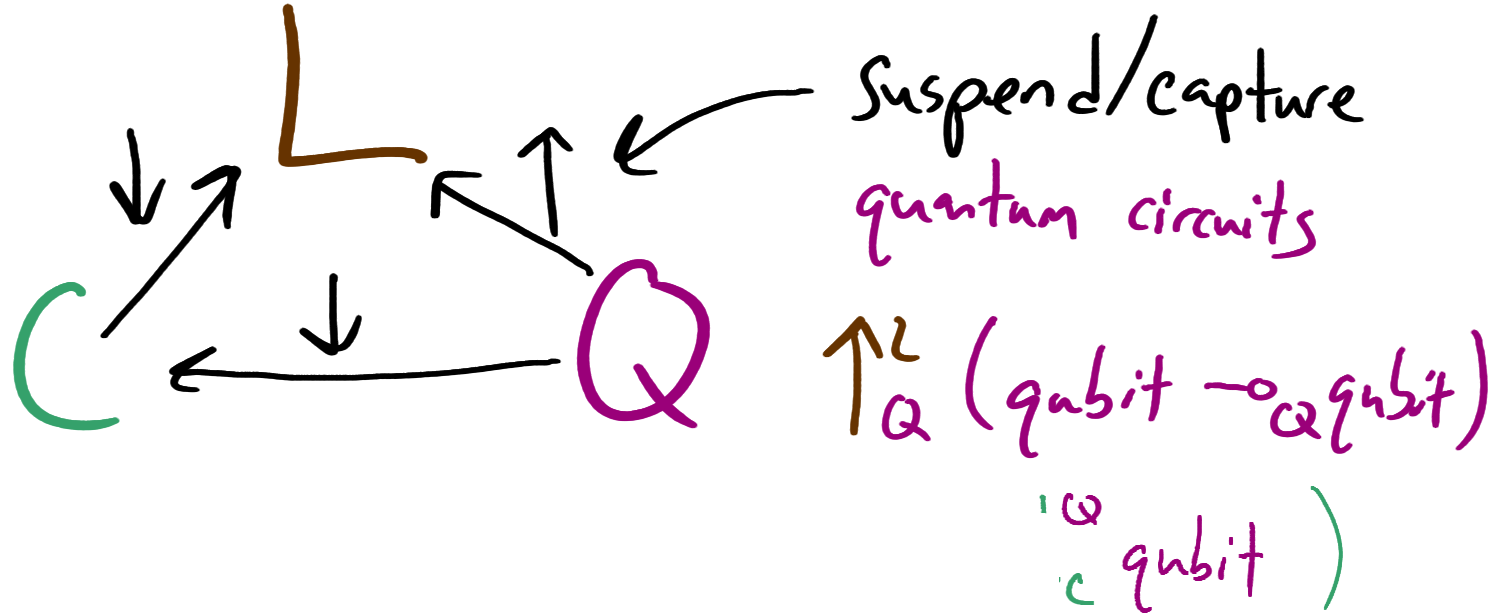


Q
C qubit)

HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

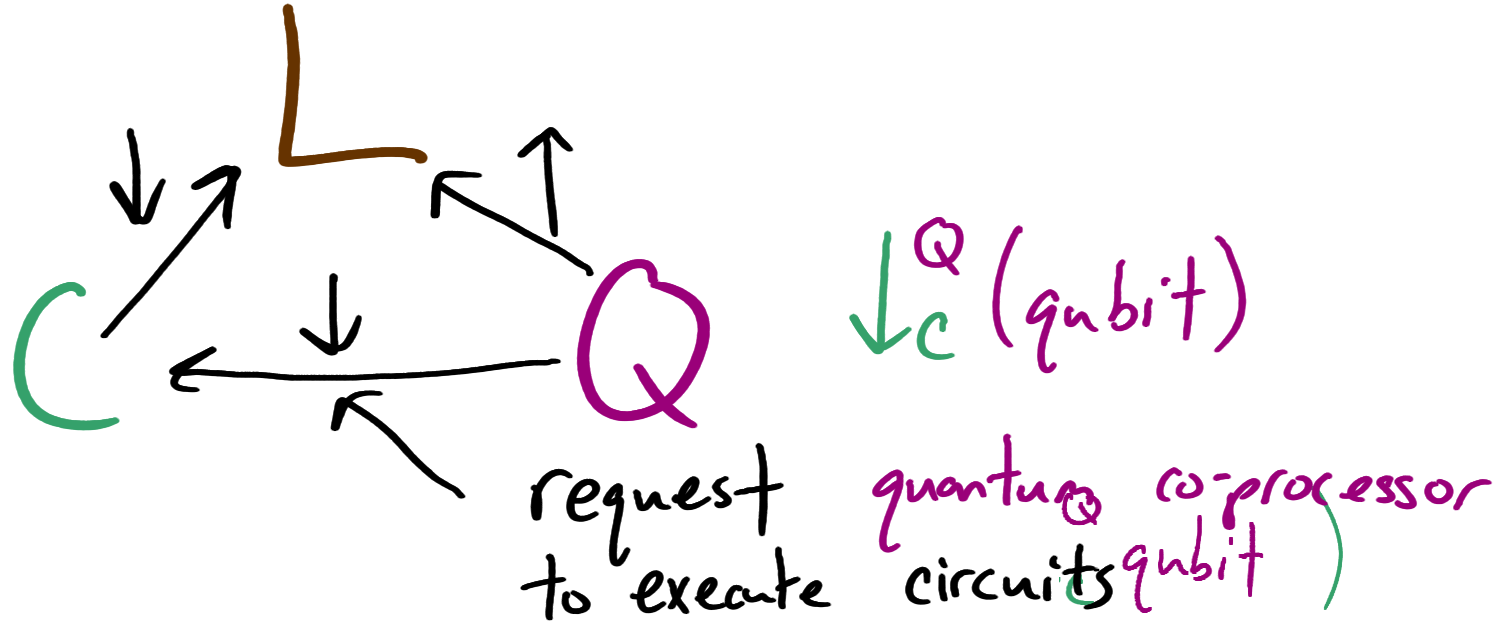
Meta programming Hybrid Programs



HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

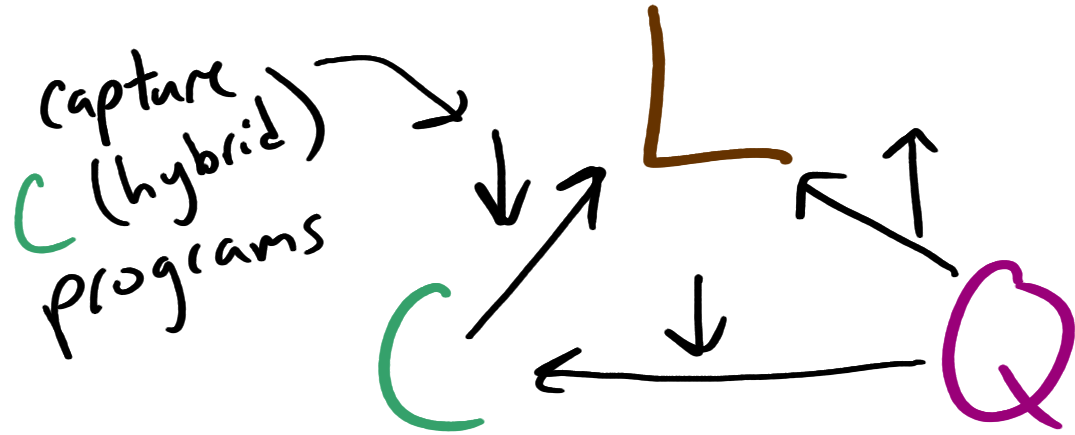
Meta programming Hybrid Programs



HyQ: Staged Hybrid Quantum-Classical Programming

Chuta Sano

Meta programming Hybrid Programs



down_L^C (teleport): $\downarrow_L^C (\downarrow_c^Q \text{ qubit} \rightarrow_c \downarrow_c^Q \text{ qubit})$

Staging

- Stage 1: Perform all $\angle$ reductions
(i.e. normalize/pre-generate all code)
- Stage 2: Run hybrid (C) program

Stage 1: Circuit Normalization

-Key Idea: Reduce under binders to find
 $\hookrightarrow$ reductions

$$\frac{M_m \mapsto M'_m \quad (\text{where } m \in \{Q, C\})}{\lambda x.M_m \mapsto \lambda x.M'_m}$$

$$\frac{N \mapsto N' \quad (\text{where } m \in \{Q, C\})}{\text{let } \langle x, y \rangle_m = R \text{ in } N \mapsto \text{let } \langle x, y \rangle_m = R \text{ in } N'}$$

Stage 1: Circuit Normalization

- Type Safety

- Main Theorem: If $\vdash_L M : \downarrow_L^C A_c$, then

$M = \text{down}_L^C(N_c)$ where N_c does not contain any L -subterms (or $M \rightarrow M'$)

Stage 2: Runtime Semantics

- Defined for (normalized) C/Q ... "standard"
- Type Safety

Conclusion

- A formal language that (provably) pre-generates quantum circuits in a hybrid program
 - Adjoint Logic to modularly separate three semantically different languages
 - In the paper: 5 modes, precise meaning of pre-generation
-