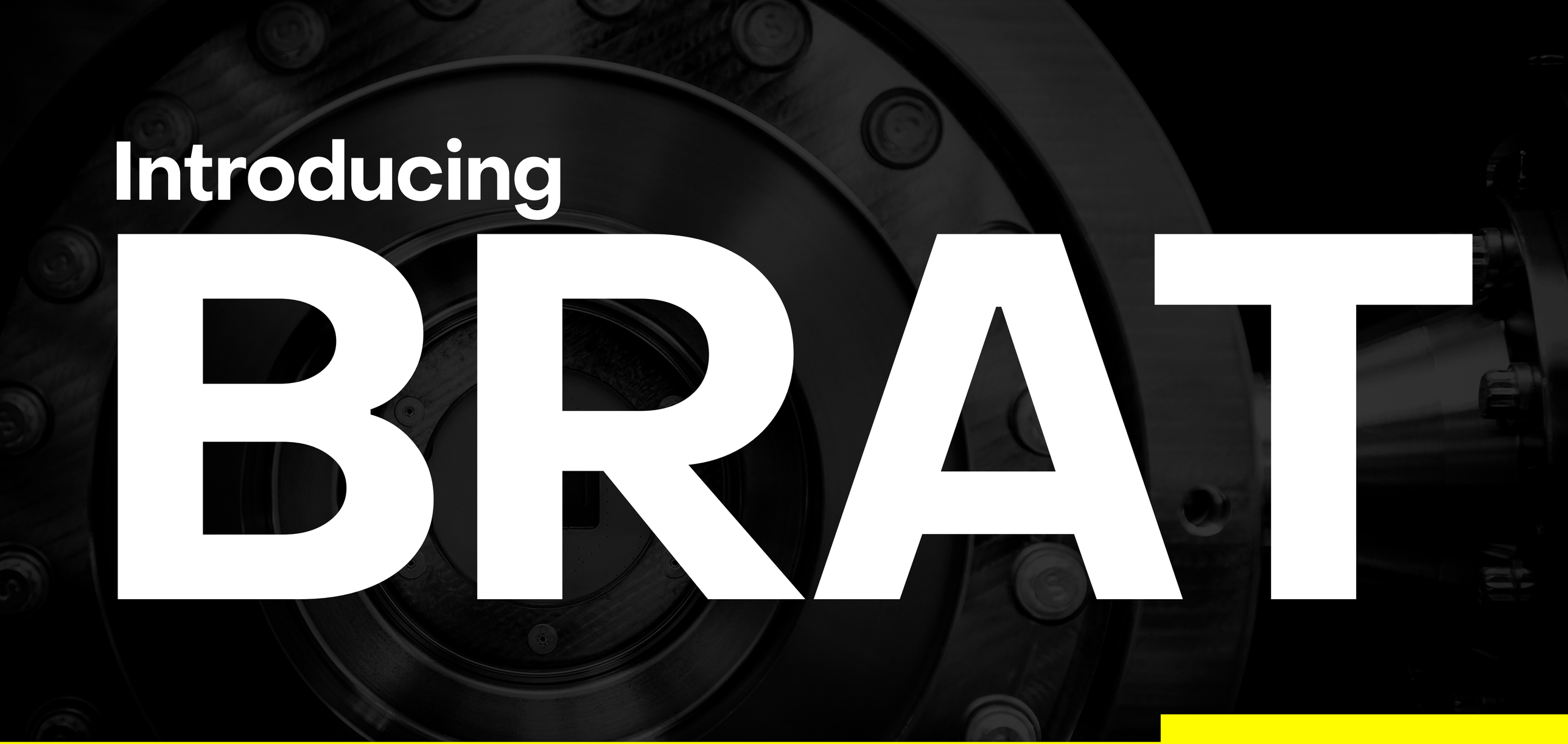
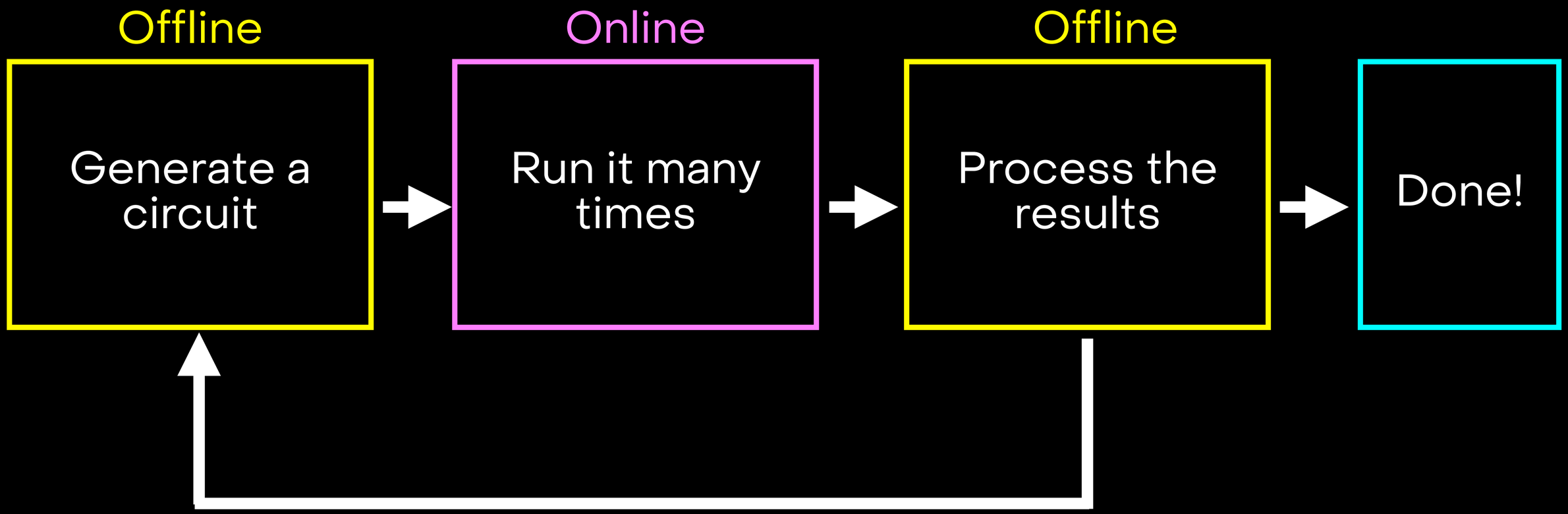


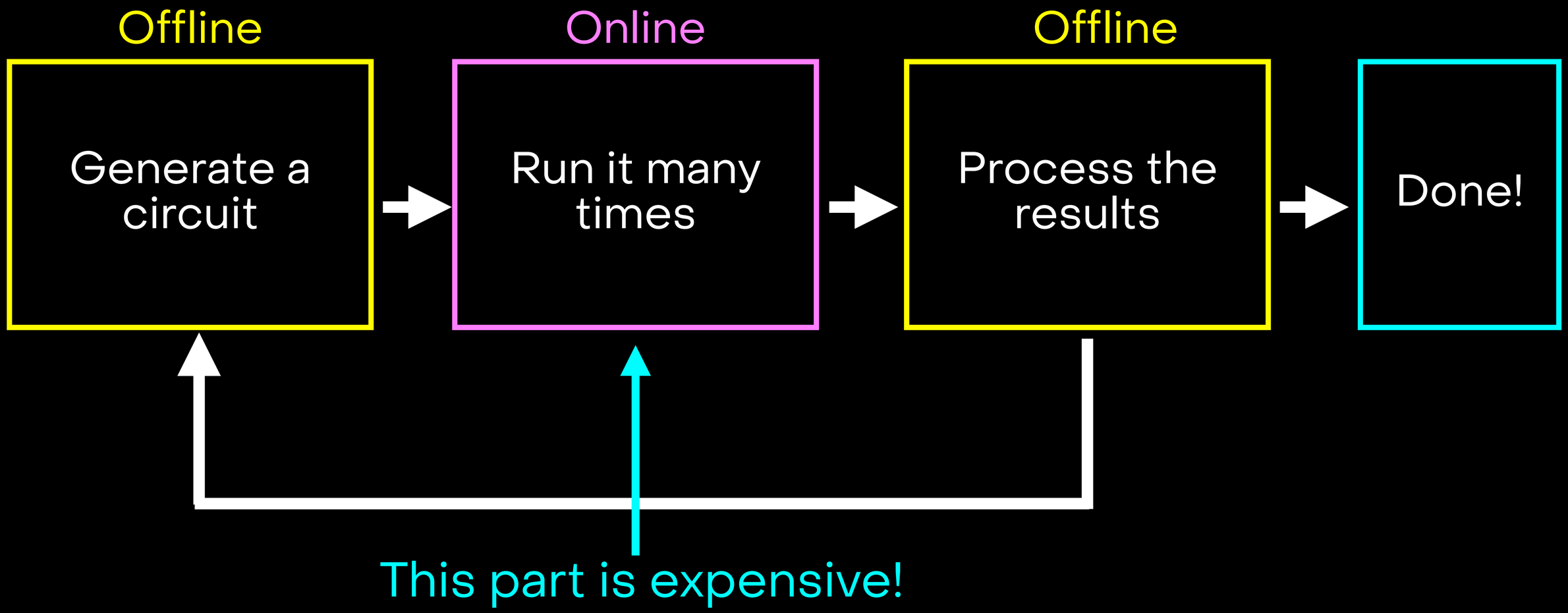
Ross Duncan, Mark Koch, Alan Lawrence, Conor McBride, **Craig Roy**

QUANTINUUM

A Typical Quantum Experiment

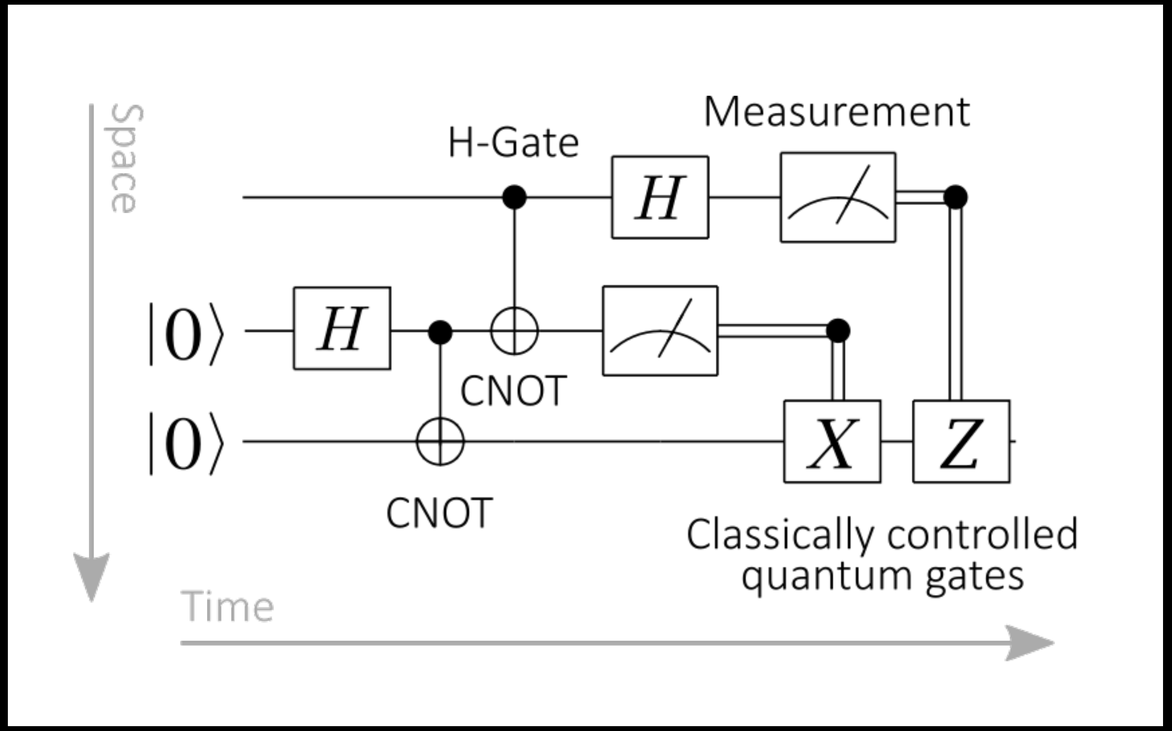


A Typical Quantum Experiment



Typical Quantum Circuits

Finite, and usually small



Qubits measured at the end, into bit strings

Entropy increasing with time

Typical Quantum States

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Adding qubits blows up the state space via tensor product:

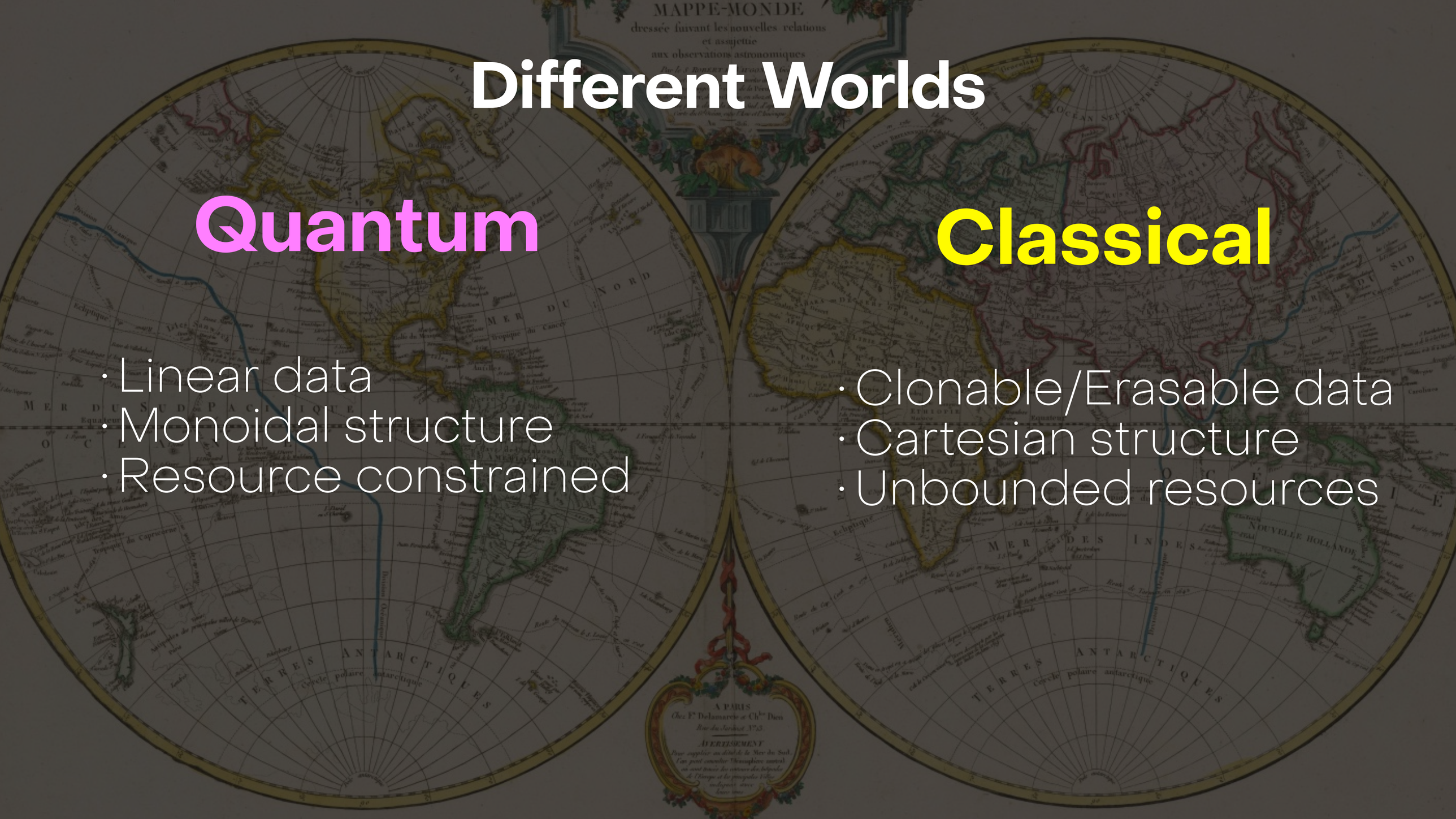
$$|\psi\phi\rangle = |\psi\rangle \otimes |\phi\rangle$$

No go theorems: Cloning; Deletion?

Quantum State Evolution

$$(H \otimes I)|\psi\phi\rangle = H|\psi\rangle \otimes |\phi\rangle$$

$$CX|\psi\phi\rangle$$



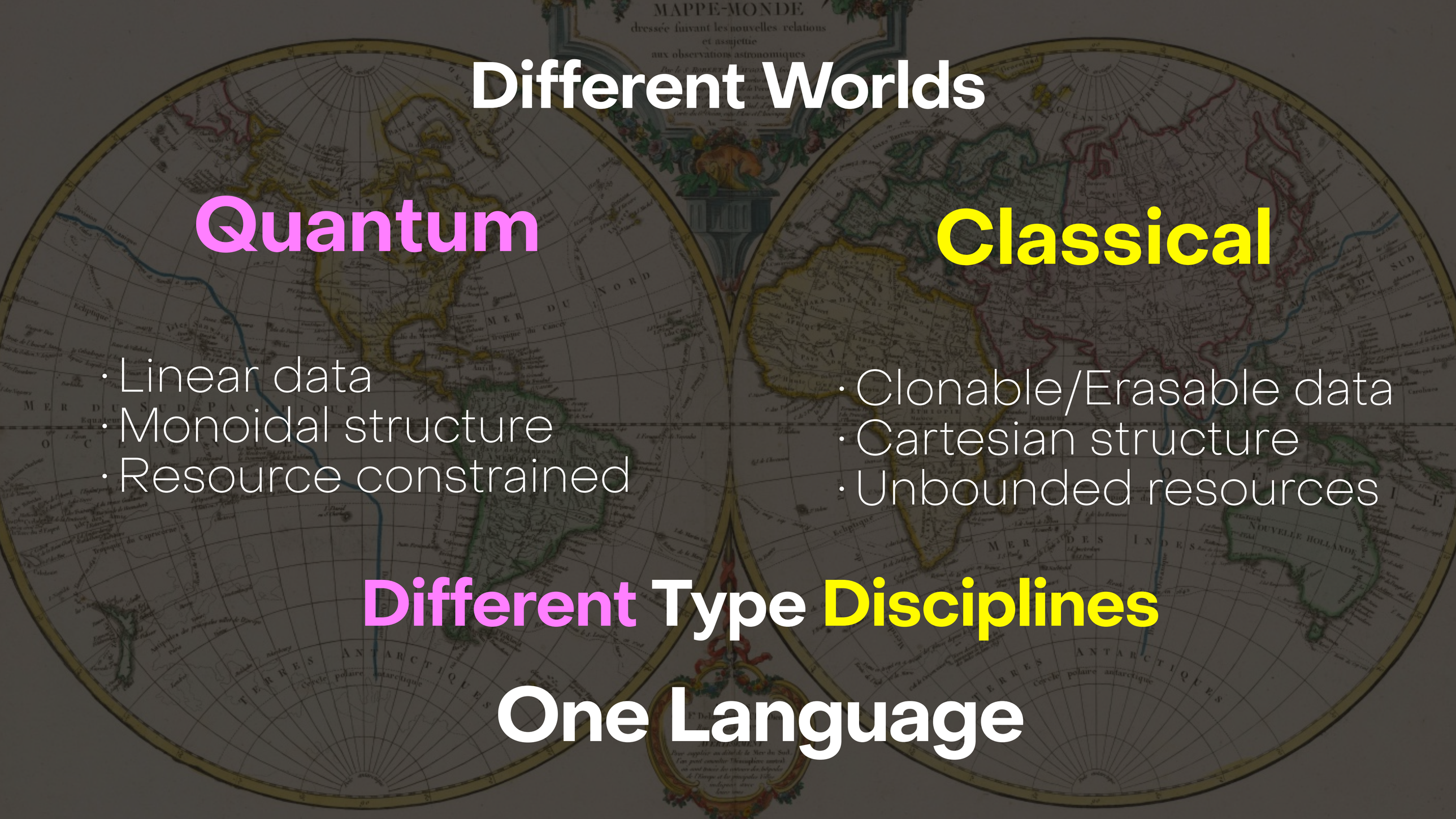
Different Worlds

Quantum

- Linear data
- Monoidal structure
- Resource constrained

Classical

- Clonable/Erasable data
- Cartesian structure
- Unbounded resources



Different Worlds

Quantum

- Linear data
- Monoidal structure
- Resource constrained

Classical

- Clonable/Erasable data
- Cartesian structure
- Unbounded resources

Different Type Disciplines

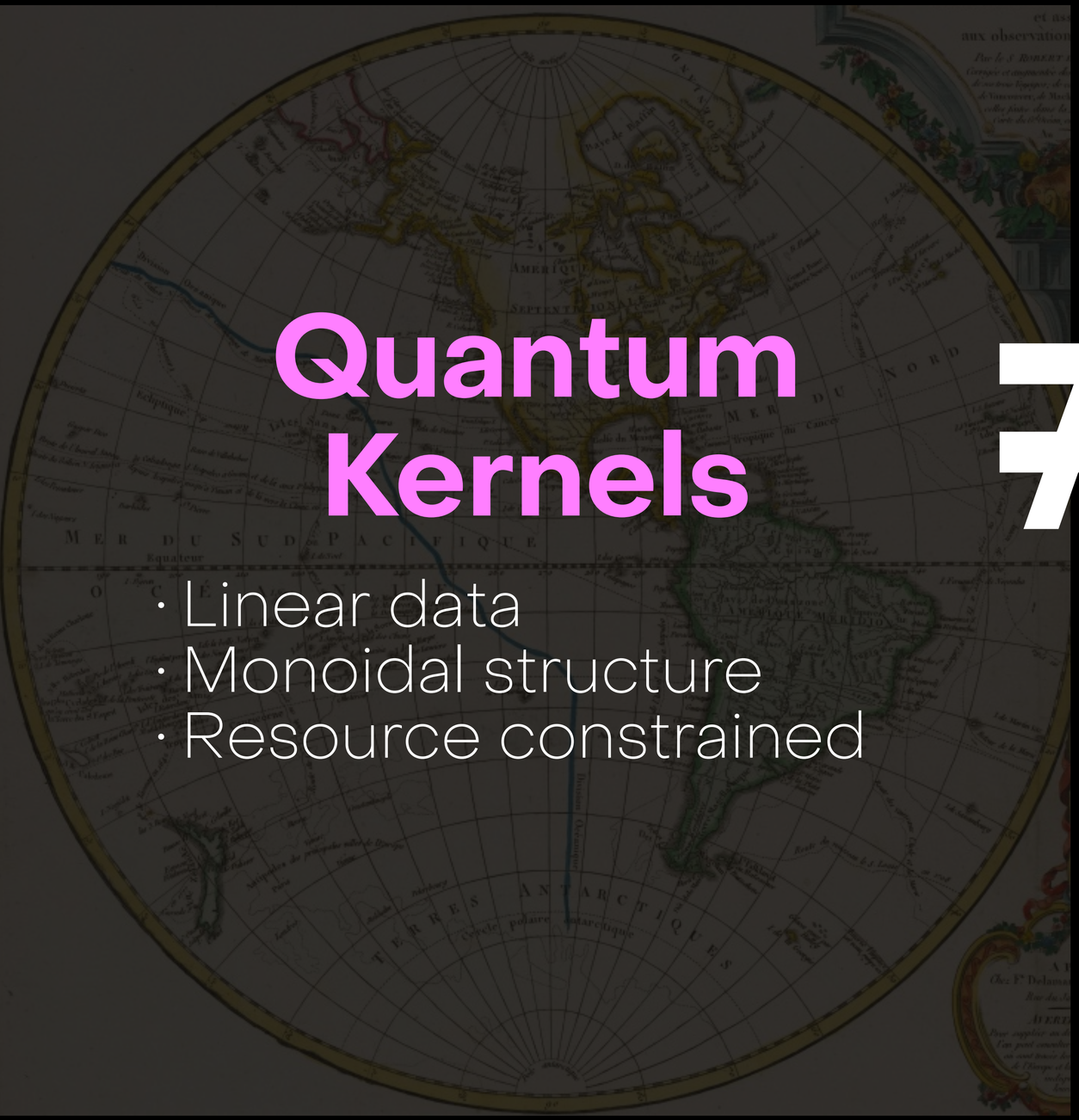
One Language

Quantum

Generates Code

Returns Data

Classical



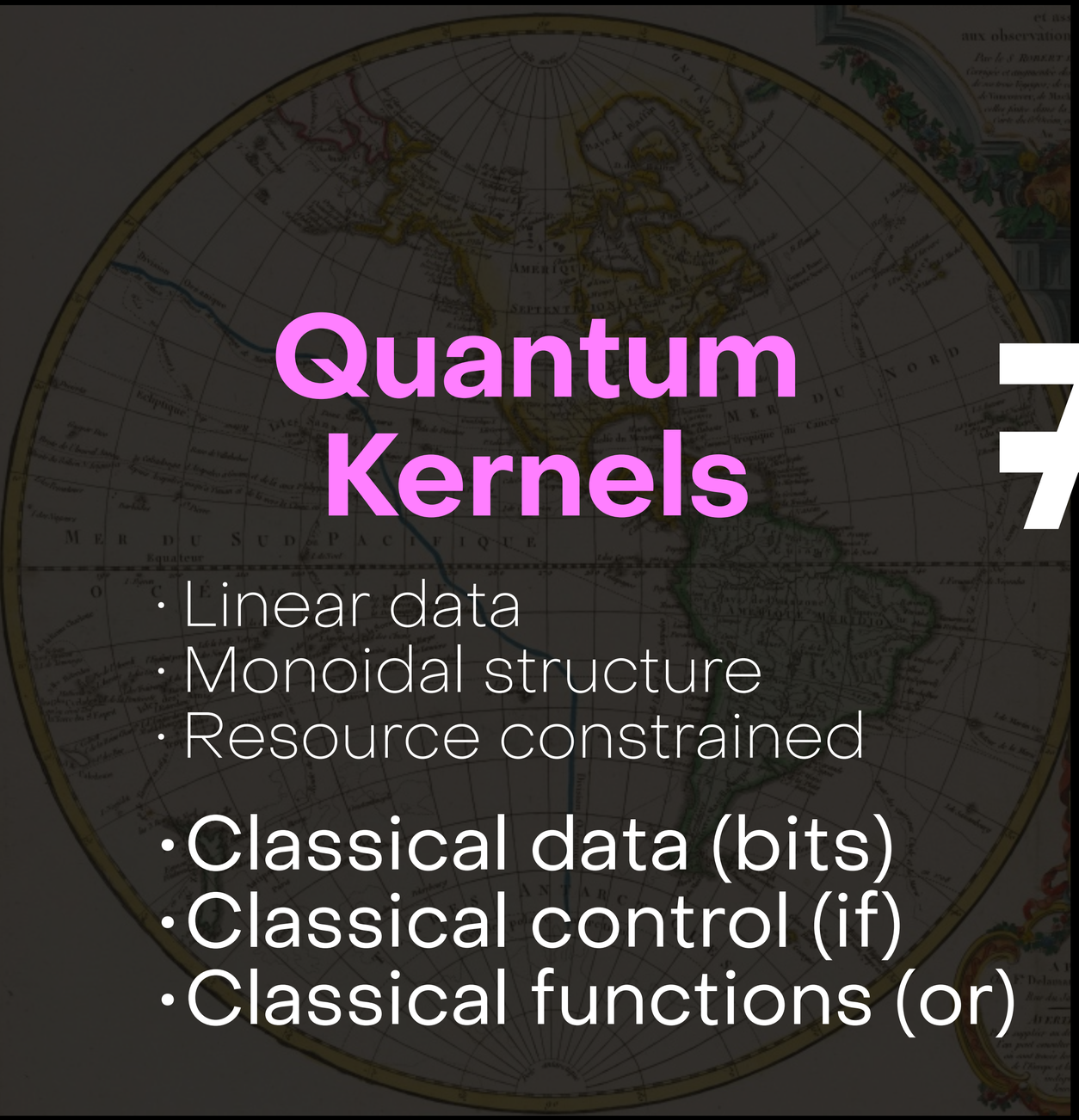
Quantum Kernels

- Linear data
- Monoidal structure
- Resource constrained

≠

Quantum Circuits

- Linear data
- Monoidal structure
- Resource constrained



Quantum Kernels

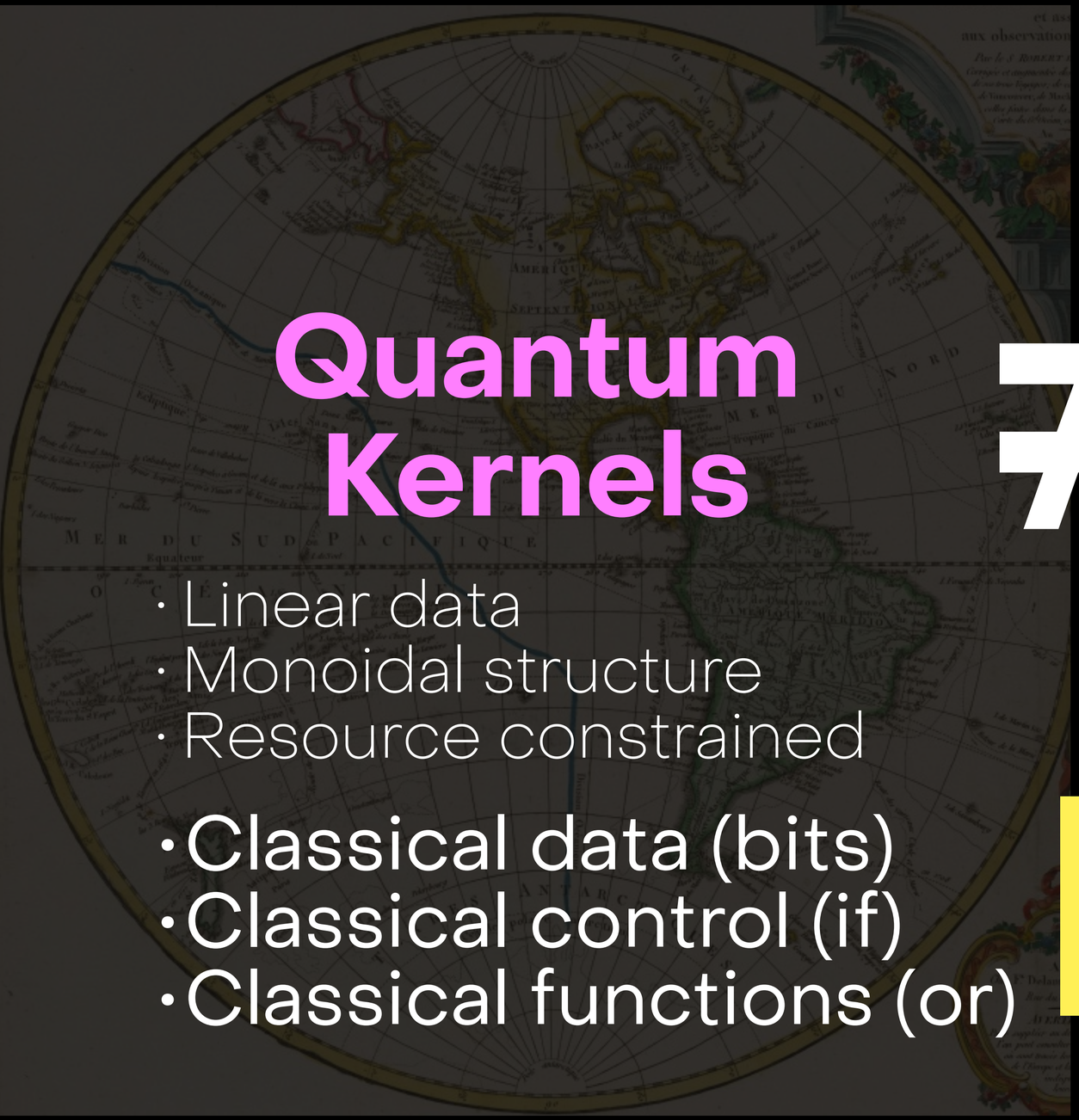
- Linear data
- Monoidal structure
- Resource constrained

- Classical data (bits)
- Classical control (if)
- Classical functions (or)

≠

Quantum Circuits

- Linear data
- Monoidal structure
- Resource constrained



Quantum Kernels

- Linear data
- Monoidal structure
- Resource constrained

- Classical data (bits)
- Classical control (if)
- Classical functions (or)

≠

Quantum Circuits

- Linear data
- Monoidal structure
- Resource constrained

Device Dependent

Programming in two worlds

```
fst(Qubit, Qubit) -o Qubit
fst(a, b) = a
```

```
fst(Int, Int) -> Int
fst(a, b) = a
```

Programming in two worlds

```
fst(Qubit, Qubit) -o Qubit
fst(a, b) = a
```

```
Type error: Variable(s) b haven't been used
```

```
fst(Int, Int) -> Int
fst(a, b) = a
```

Programming in two worlds

```
copy(Qubit) -o Qubit, Qubit
copy(q) = q,q
```

```
copy(Bool) -> Bool, Bool
copy(a) = a, a
```

Programming in two worlds

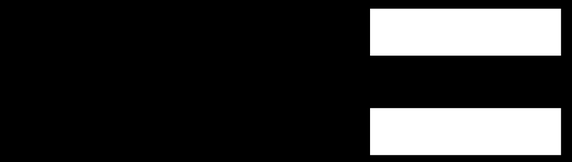
```
copy(Qubit) -o Qubit, Qubit
copy(q) = q,q
```

```
Type error: q has already been used
```

```
copy(Bool) -> Bool, Bool
copy(a) = a, a
```

Intro to BRAT syntax

```
circ(Qubit, Qubit) -o Qubit, Qubit
```



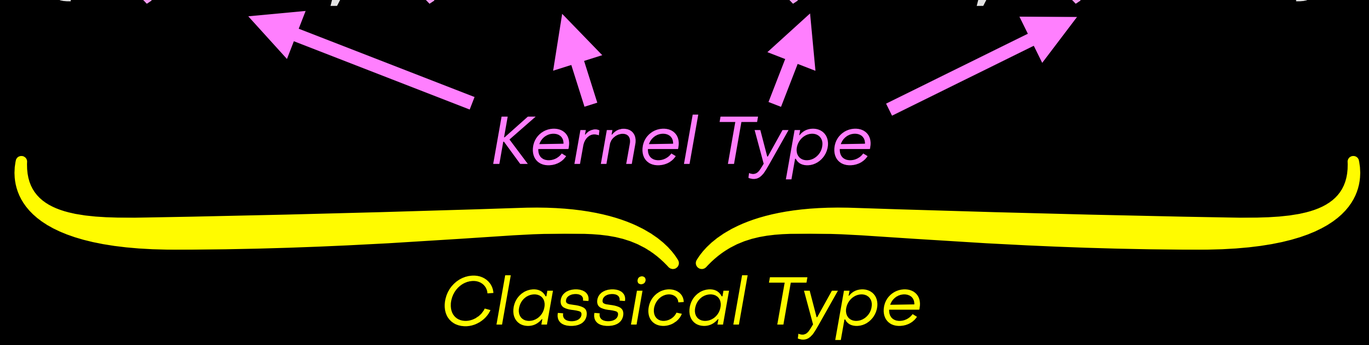
```
circ :: { Qubit, Qubit -o Qubit, Qubit }
```

Intro to BRAT syntax

```
circ(Qubit, Qubit) -o Qubit, Qubit
```

=

```
circ :: { Qubit, Qubit -o Qubit, Qubit }
```



Compositional Syntax

```
open import lib.kernel

circ(Qubit, Qubit) -o Qubit, Qubit
circ = {
  H, qid;
  T, qid;
   CX;
  S, T
}
```

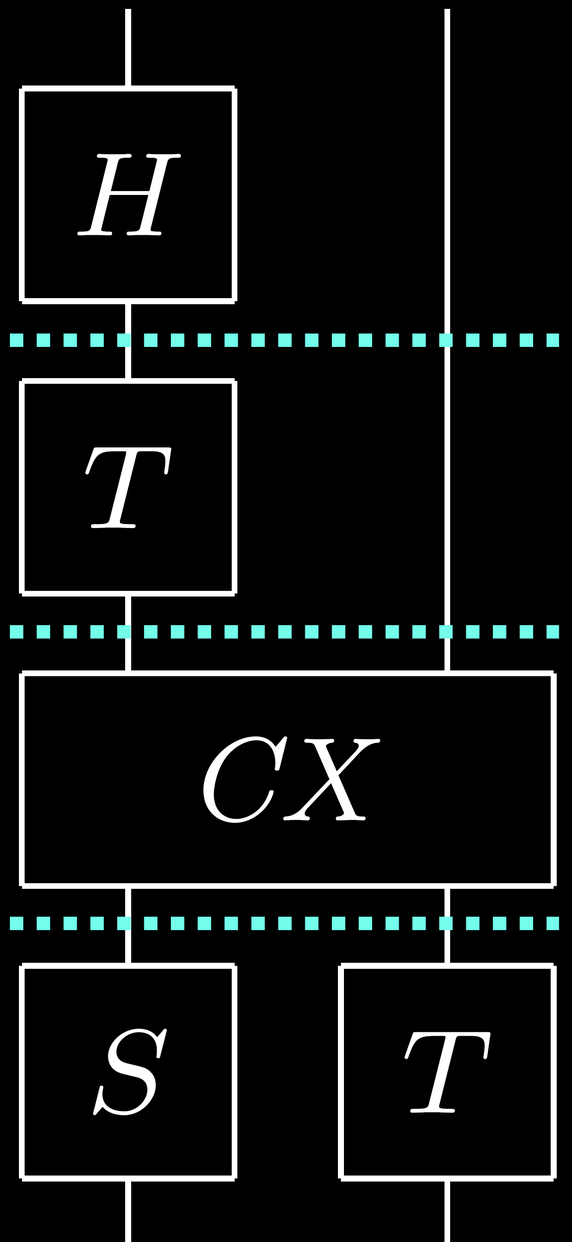
Kernel

Classical thunk

Compositional Syntax

```
open import lib.kernel

circ(Qubit, Qubit) -o Qubit, Qubit
circ = {
  H, qid;
  T, qid;
   CX;
  S, T
}
```

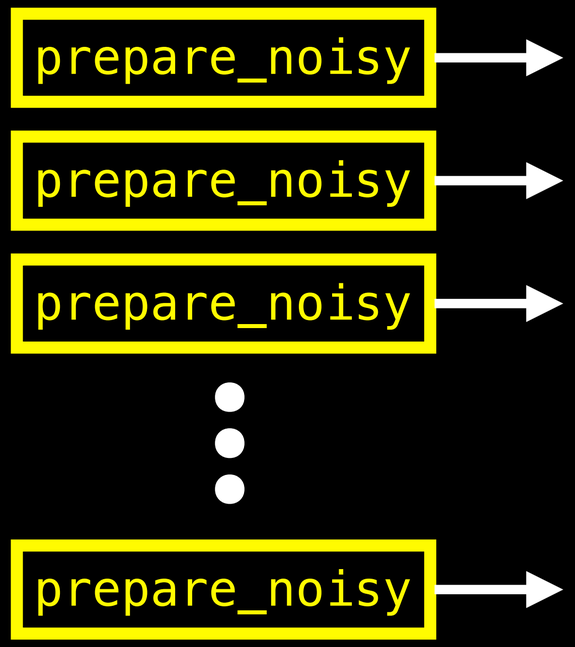


Example : Magic State Distillation

Make one good T state from a bunch of crappy ones

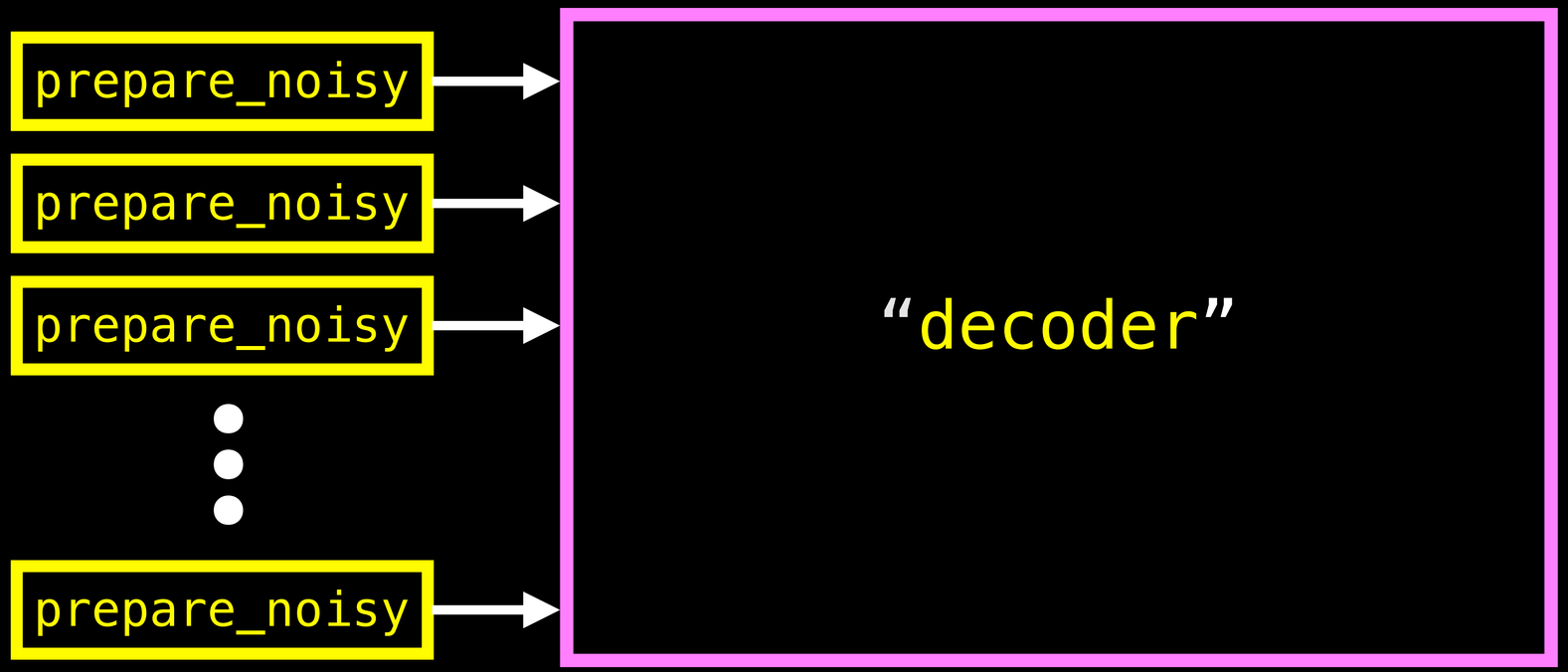
Example : Magic State Distillation

Make one good T state from a bunch of crappy ones



Example : Magic State Distillation

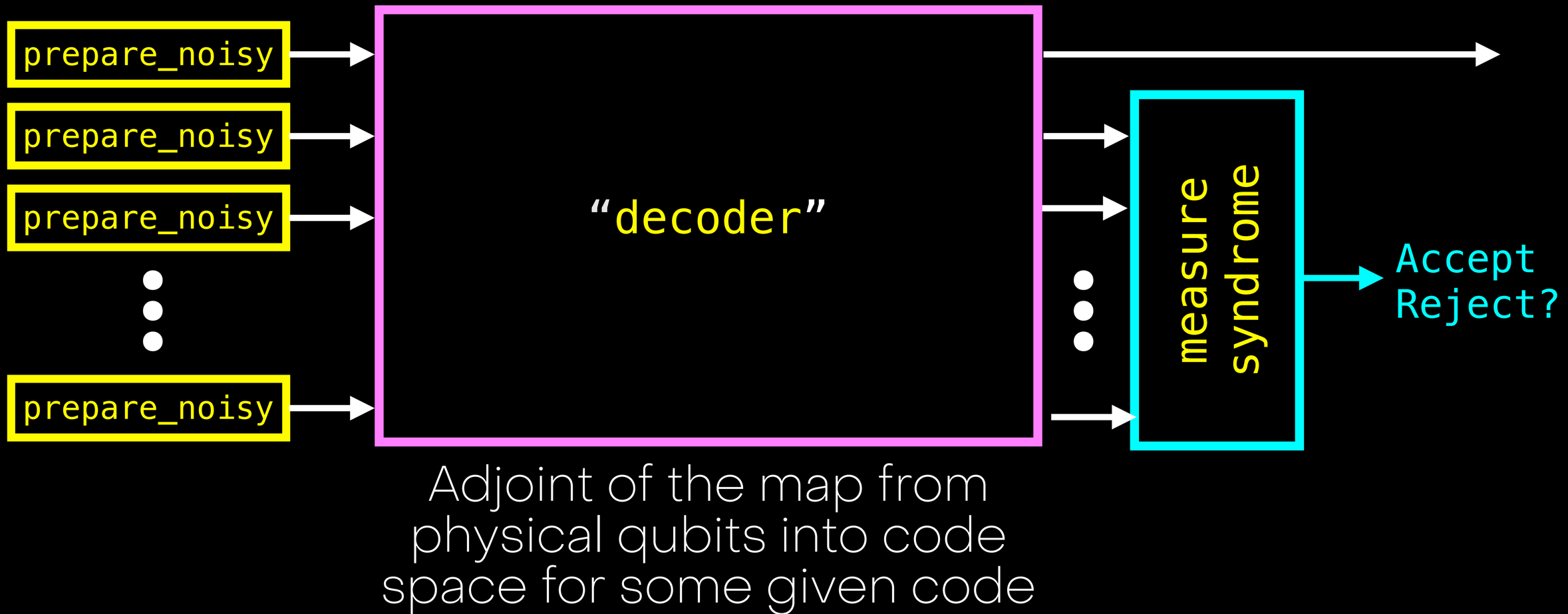
Make one good T state from a bunch of crappy ones



Adjoint of the map from physical qubits into code space for some given code

Example : Magic State Distillation

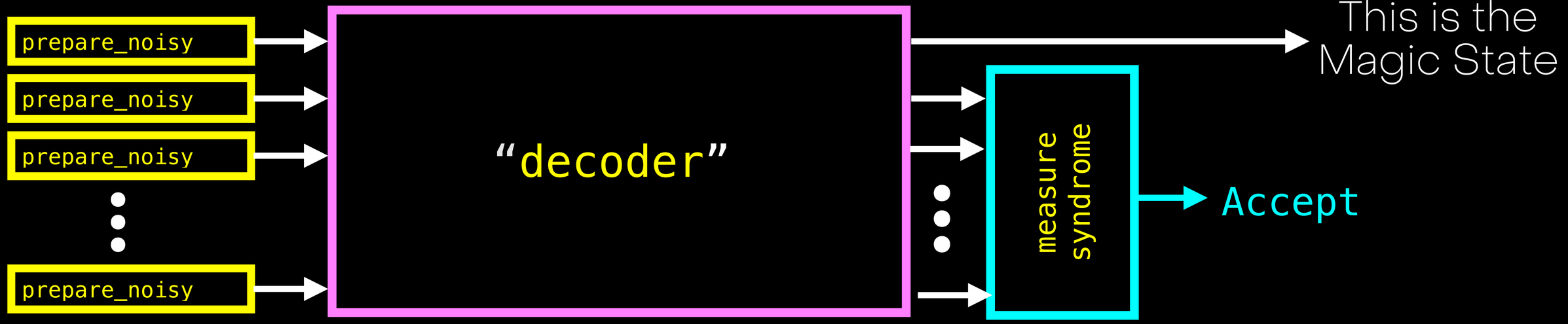
Make one good T state from a bunch of crappy ones



Adjoint of the map from physical qubits into code space for some given code

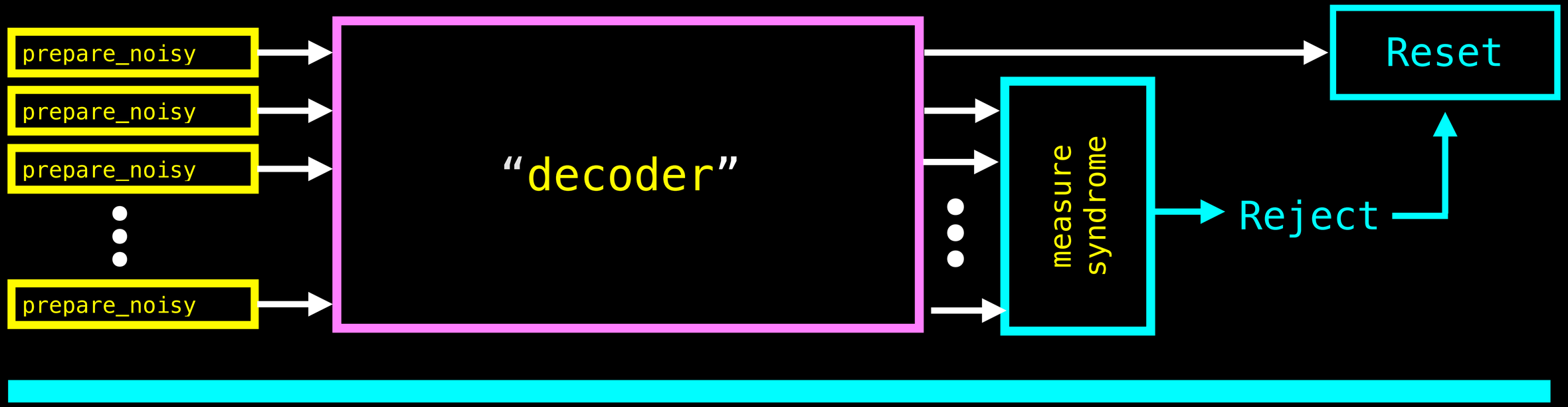
Example : Magic State Distillation

Make one good T state from a bunch of crappy ones



Example : Magic State Distillation

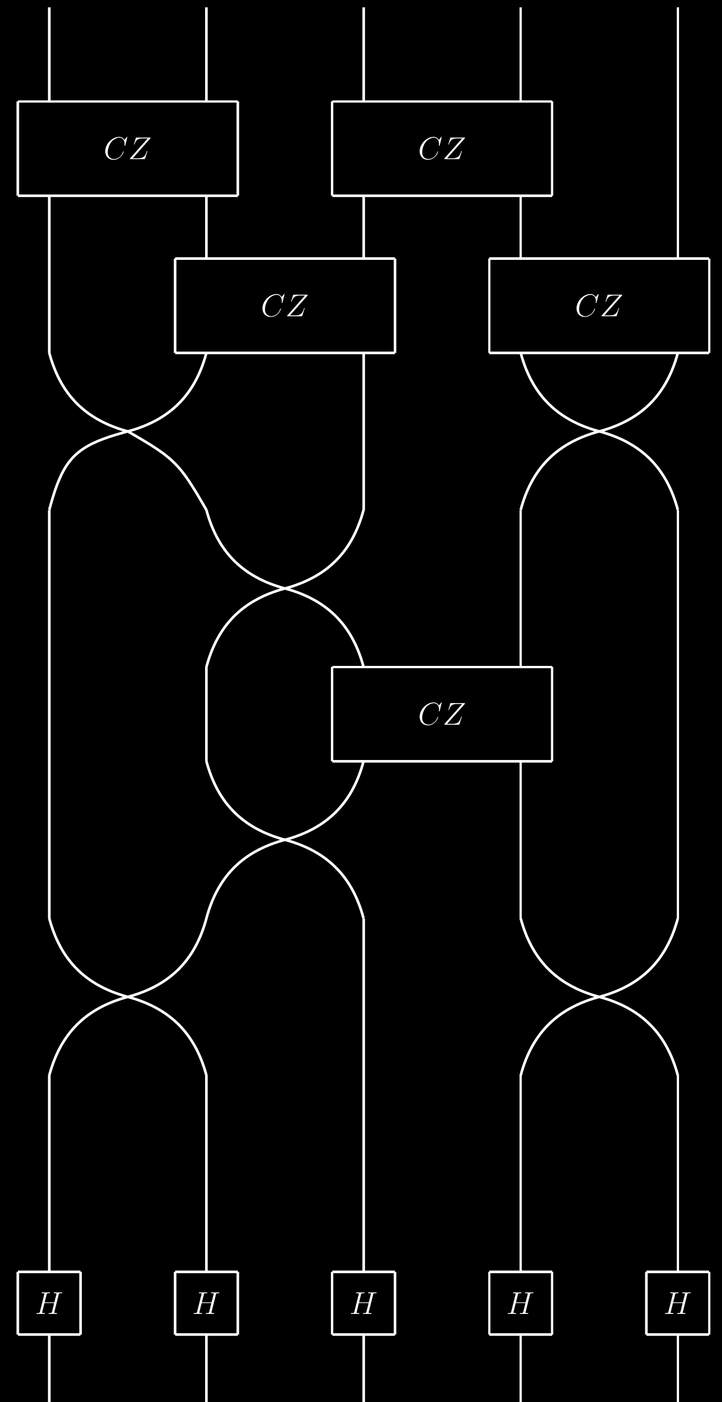
Make one good T state from a bunch of crappy ones



Then do the whole thing again

Decoder circuit

```
decoder(Vec(Qubit, 5)) -o Vec(Qubit, 5)
decoder = {
  [/\];
  CZ, CZ, qid;
  qid, CZ, CZ;
  swap, qid, swap;
  qid, swap, qid, qid;
  qid, qid, CZ, qid;
  qid, swap, qid, qid;
  swap, qid, swap;
  H, H, H, H, H;
  [\/]
}
```



Programming in two worlds II

Polymorphism over two Kinds

```
qid(Qubit) -o Qubit
qid(q) = q
```

```
generic_id(X :: *) -> { X -> X }
generic_id(_) = { x => x }
```

Programming in two worlds II

Polymorphism over two Kinds

```
qid(Qubit) -o Qubit
qid(q) = q
```

```
generic_id(X :: *) -> { X -> X }
generic_id(_) = { x => x }
```

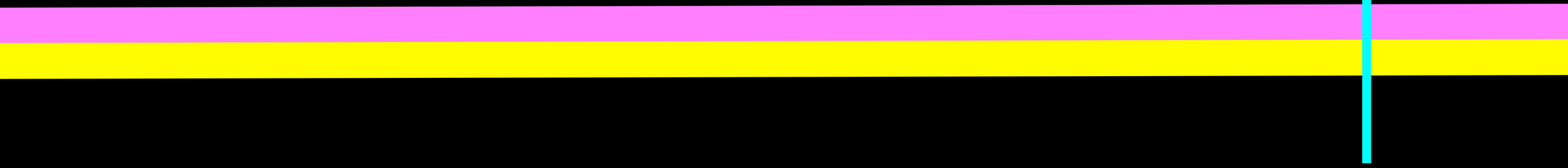
```
kid(X :: $) -> { X -o X }
kid(_) = { x => x }
```

Programming in two worlds II

Polymorphism over two Kinds

```
qid(Qubit) -> Qubit
qid(q) = q
```

```
qid(Qubit) -o Qubit
qid = kid(Qubit)
```



```
generic_id(X :: *) -> { X -> X }
generic_id(_) = { x => x }
```

```
kid(X :: $) -> { X -o X }
kid(_) = { x => x }
```

Classic functional programs

```
map(X :: $, Y :: $, { X -o Y }, n :: #)
    ->
    { Vec(X, n) -o Vec(Y, n) }

map(_, _, _, 0)          = { [] => [] }
map(X, Y, f, succ(n)) =
  { cons(x, xs) => cons(f(x), map(X, Y, f, n)(xs)) }
```

Decoder circuit

```
decoder(Vec(Qubit, 5))
        -o Vec(Qubit, 5)
decoder = {
  [/\];
  CZ, CZ, qid;
  qid, CZ, CZ;
  swap, qid, swap;
  qid, qid, CZ, qid;
  swap, qid, swap;
  H, H, H, H, H;
  [\/]
}
```

```
decoder(Vec(Qubit, 5))
        -o Vec(Qubit, 5)
decoder = {
  [/\];
  CZ, CZ, qid;
  qid, CZ, CZ;
  swap, qid, swap;
  qid, qid, CZ, qid;
  swap, qid, swap;
  [\/];
  map(Qubit, Qubit, H, 5)
}
```

Classic functional programs II

```
fold(X :: $
    ,{ X, X -o X }
    ,{ -o X }
    ,n :: #
    ) -> { Vec(X, n) -o X }
```

```
any(n :: #) -> { Vec(Bit, n) -o Bit }
any(n) = fold(Bit, or, { => false }, n)
```

Syndrome Measurement

```
measure_all(n :: #)
        -> { Vec(Qubit, n)
          -o Vec(Bit, n), Vec(Money, n)
           }

measure_syndrome(n :: #)
              -> { Vec(Qubit, n)
                -o Bit, Vec(Money, n)
                 }
measure_syndrome(n) = {
  measure_all(n);
  any(n), kid(Vec(Money, n))
}
```

Money

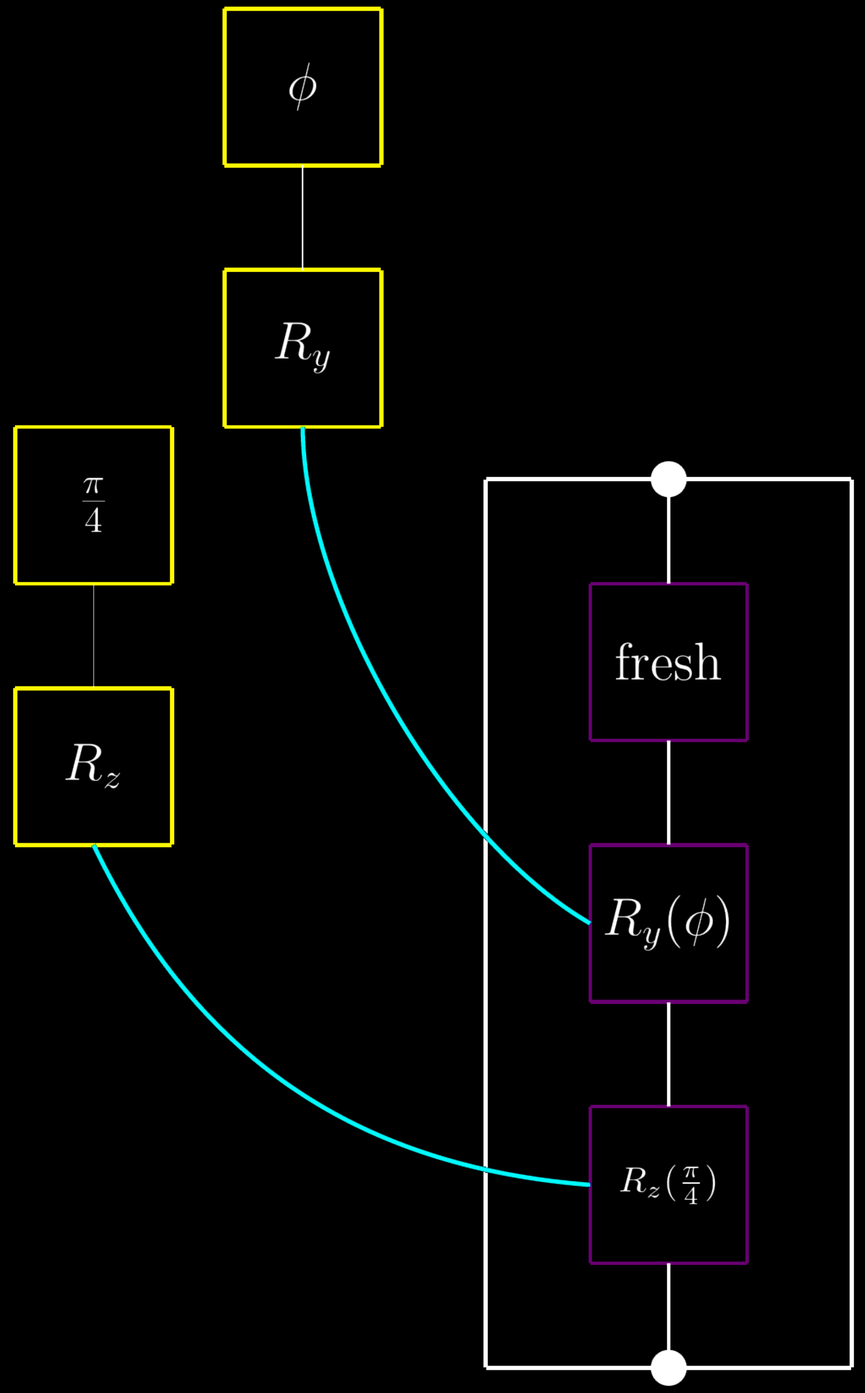
```
fresh(Money) -o { Qubit }

measure(Qubit) -o { Money, Bit }
```

Prep circuit

```
phi :: Float
phi = arccos(1.0 / sqrt(3.0))

prepare_noisy_1(Money) -o Qubit
prepare_noisy_1 = {
  fresh;
  Ry(phi);
  Rz(pi / 4.0)
}
```



Prep circuit

```
phi :: Float
phi = arccos(1.0 / sqrt(3.0))

prepare_noisy_1(Money) -o Qubit
prepare_noisy_1 = {
  fresh;
  Ry(phi);
  Rz(pi / 4.0)
}

prepare_noisy(n :: #) -> { Vec(Money, n) -o Vec(Qubit, n) }
prepare_noisy(n) = map(Money, Qubit, prepare_noisy_1, n)
```

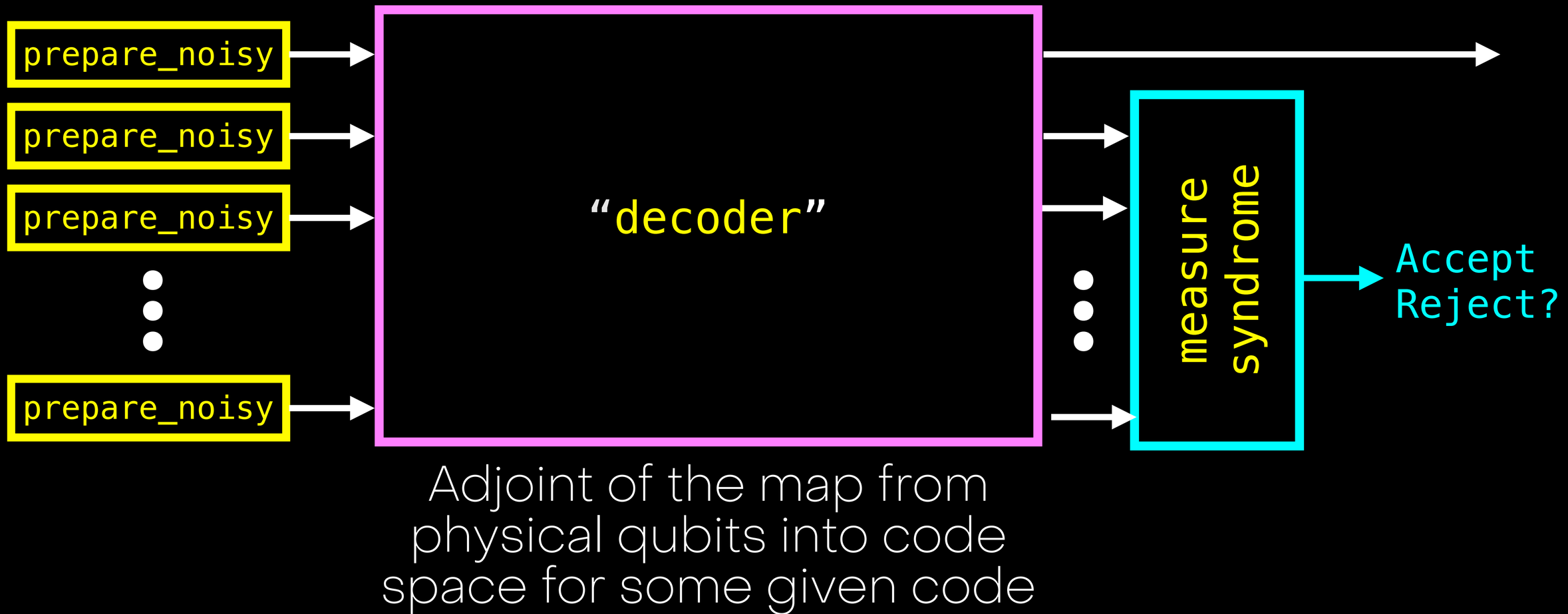
Putting it together

```
distill(Vec(Money, 5)) -o Qubit, Vec(Money, 4)
```

```
redistill_if_bad(Qubit, Bit, Vec(Money, 4))
                 -o Qubit, Vec(Money, 4)
redistill_if_bad(q, false, ms) = q, ms
redistill_if_bad(q, true, ms) = let m, _ = measure(q) in
                                distill(cons(m, ms))
```

Example : Magic State Distillation

Make one good T state from a bunch of crappy ones



Adjoint of the map from physical qubits into code space for some given code

Putting it together

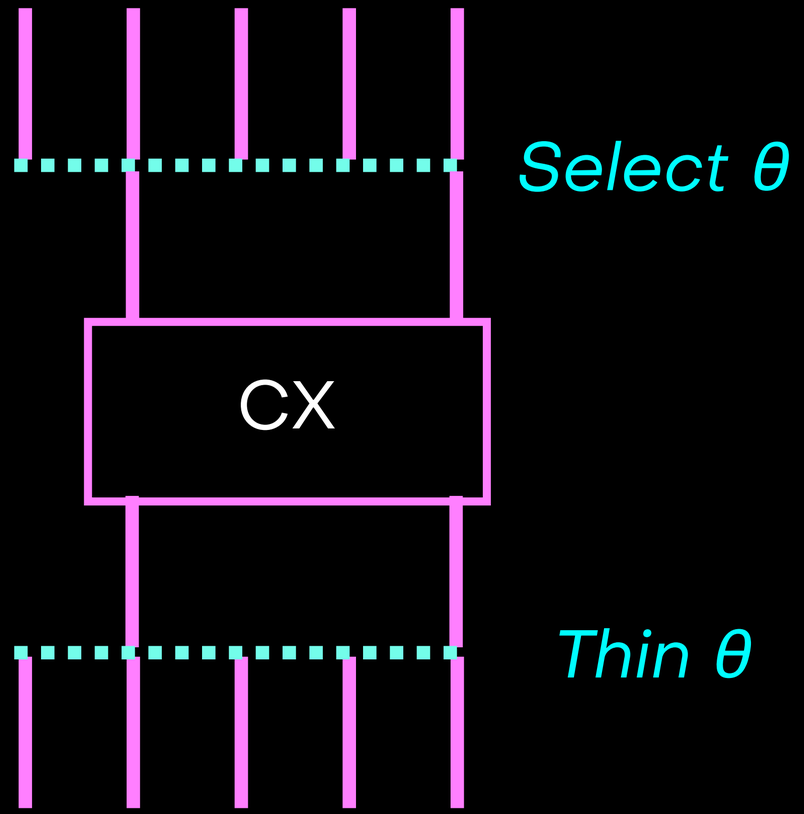
```
distill(Vec(Money, 5)) -o Qubit, Vec(Money, 4)
distill = {
  prepare_noisy(5);
  decoder;
  uncons(4);
  qid, measure_syndrome(4);
  redistill_if_bad
}
```

What else?

- Language of arithmetic expressions with one variable
- Port pulling
- Stretchy identity
- Interactive programming

What's Next?

- Vectorisation
- Applying computations with thinning
- Type Inference
- User defined data types
- Code generation
- Release



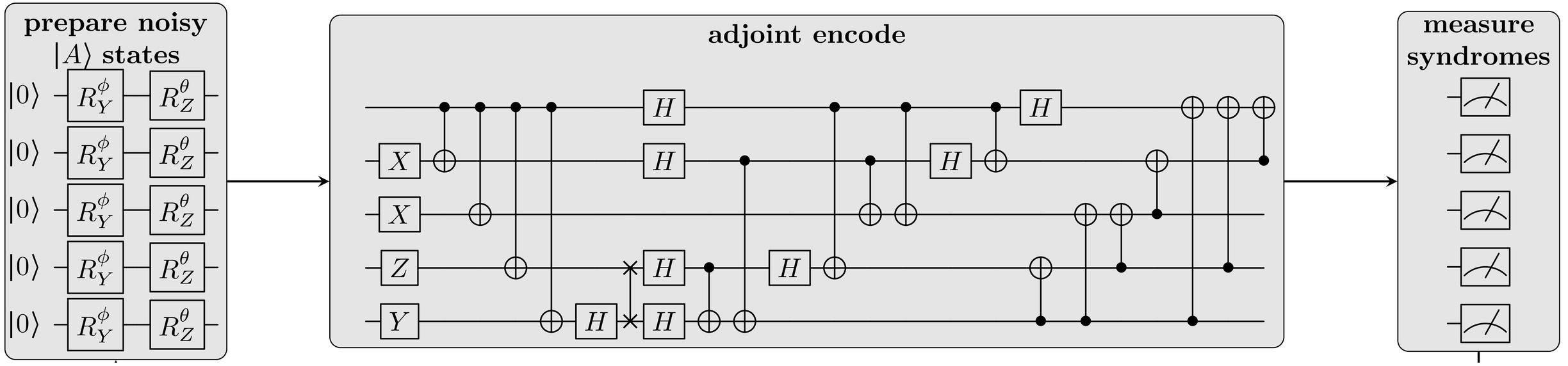
$\theta = [0,1,0,0,1]$

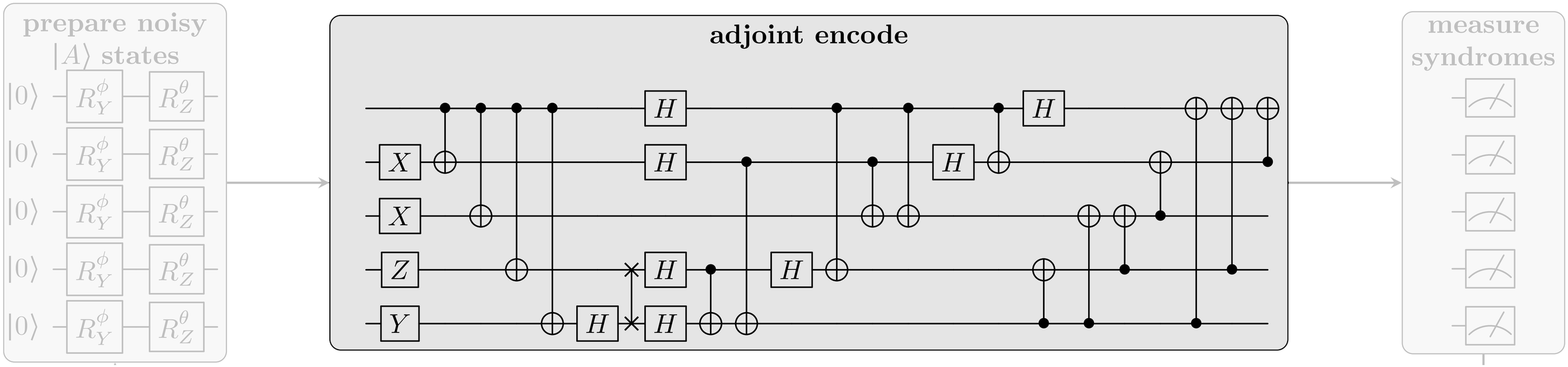
Questions?

QUANTINUUM

Advances in compilation for quantum hardware – A demonstration of magic state distillation and repeat-until-success protocols

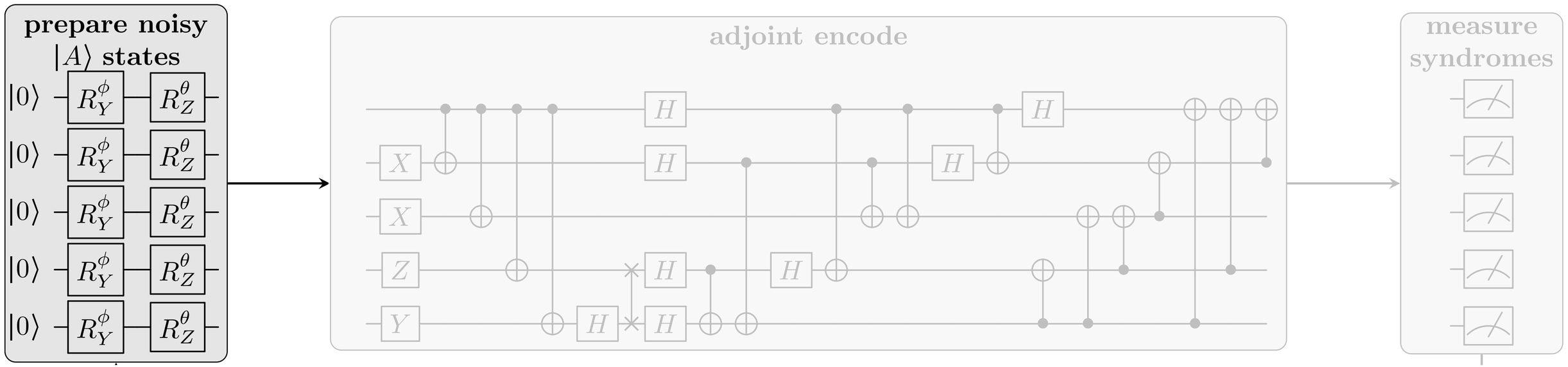
Natalie C. Brown[1], John Peter Campora III[1], Cassandra Granade[2], Bettina Heim[2], Stefan Wernli[2], Ciarán Ryan-Anderson[1], Dominic Lucchetti[1], Adam Paetznick[2], Martin Roetteler[2], Krysta Svore[2], and Alex Chernoguzov[1]



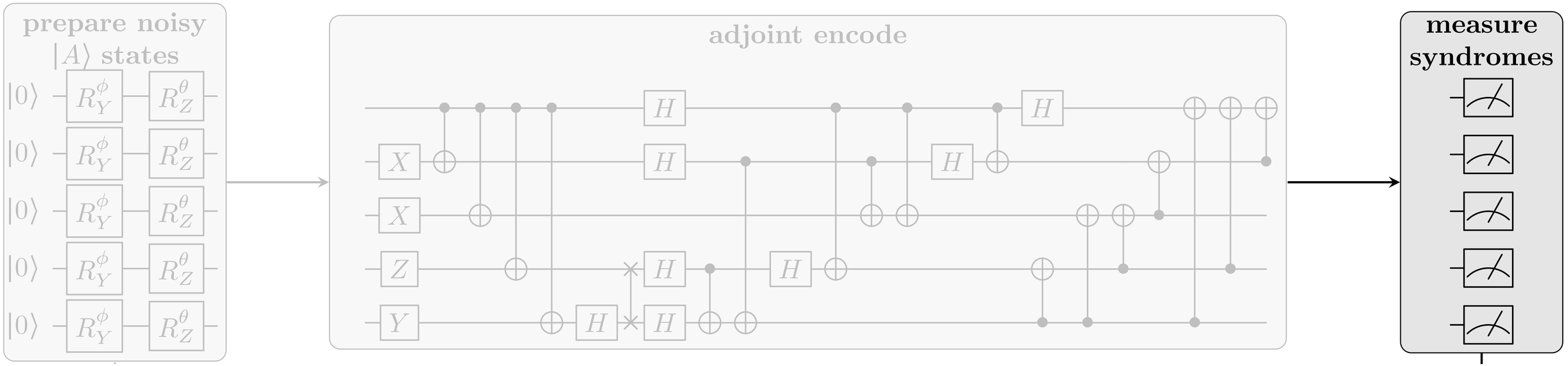


[5,1,3] code

so we're going to use 5 qubits



Simple single qubit state prep x 5



Accept = all measurements are 0

2 type universes

- Kernel types: K := Bit, Qubit, Vec(K, n)
- Type level Nats n := 0 | x | succ(n) | doub(n)
- Classical types X := x | Vec(X, n) | G | { Row(X) -> Row (X) } | { Row(K) –o Row(K) }
- Row types R := X* | K*
- Ground types G := Bool | Nat | Int | String