galois

Formal Verification of Quantum Programs

Jennifer Paykin

Pacific Northwest National Labs

October 26, 2018

About me…

- PhD from University of Pennsylvania 2018
- Now at Galois Inc in Portland OR
- Interested in programming language design, applications, & verified software
- Started working on quantum computing 2015, no physics background...

About me…

- PhD from University of Pennsylvania 2018
- Now at Galois Inc in Portland OR
- Interested in programming language design, applications, & verified software
- Started working on quantum computing 2015, no physics background...

With...

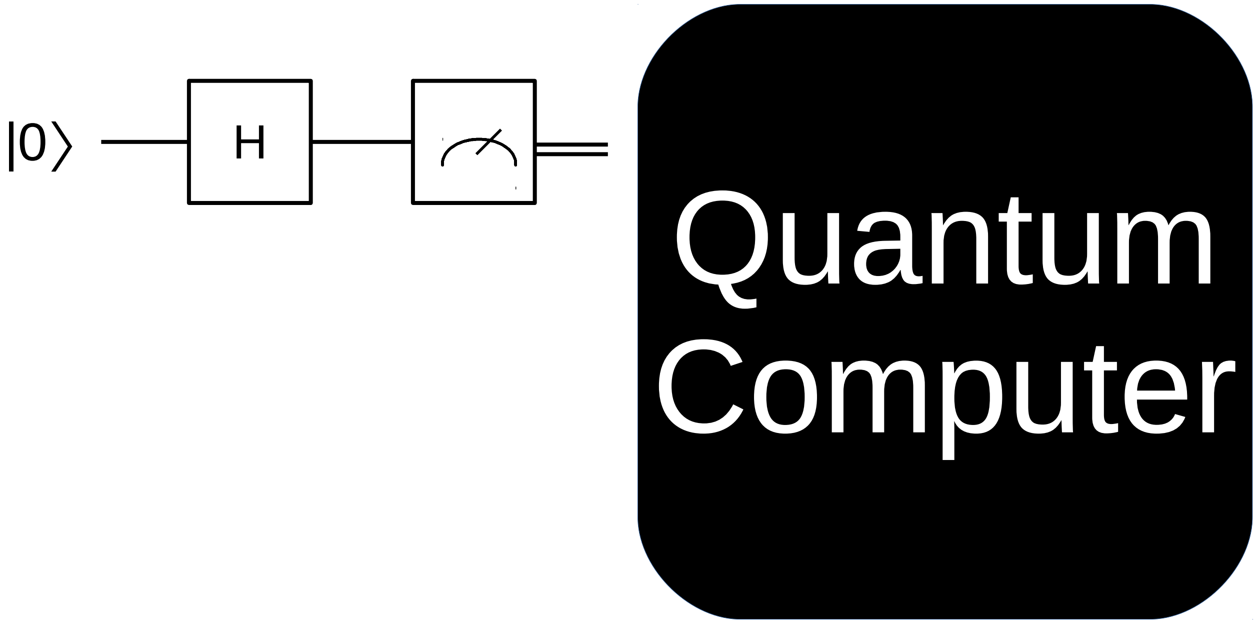
Robert Rand
University of Maryland

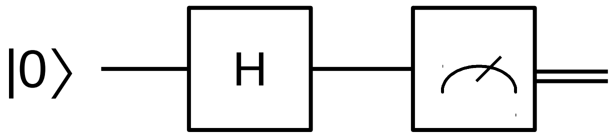
Steve Zdancewic

Dong-Ho Lee
University of Pennsylvania

Quantum Computer

Quantum
Computer


|0⟩
H
Quantum
Computer

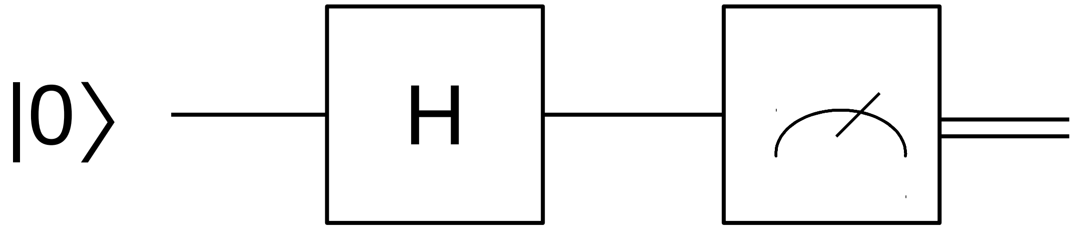


$$\begin{cases} 0 & \text{with probability 1/} \\ 1 & \text{with probability 1/} \end{cases}$$

Quantum Computer

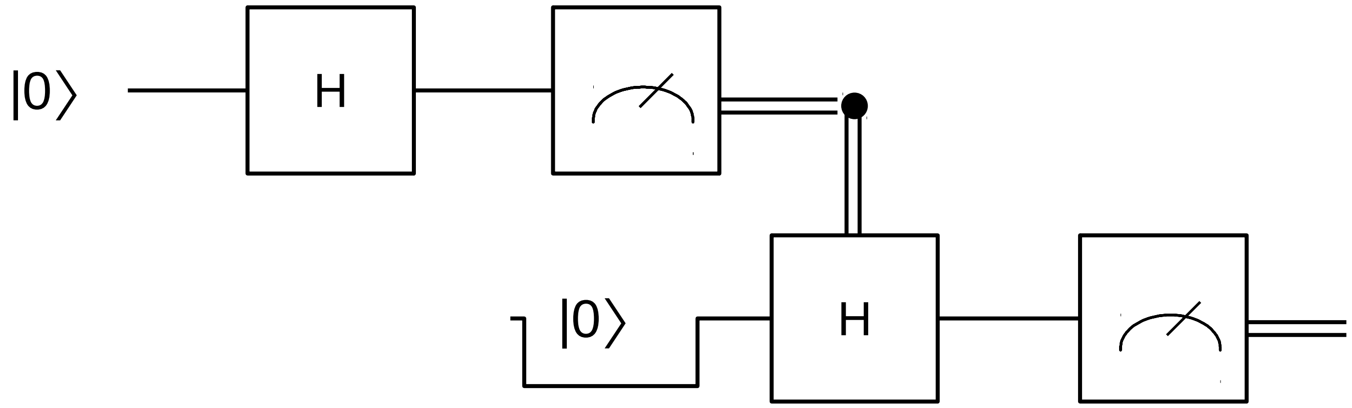
Features of Circuits

- gates applied to wires



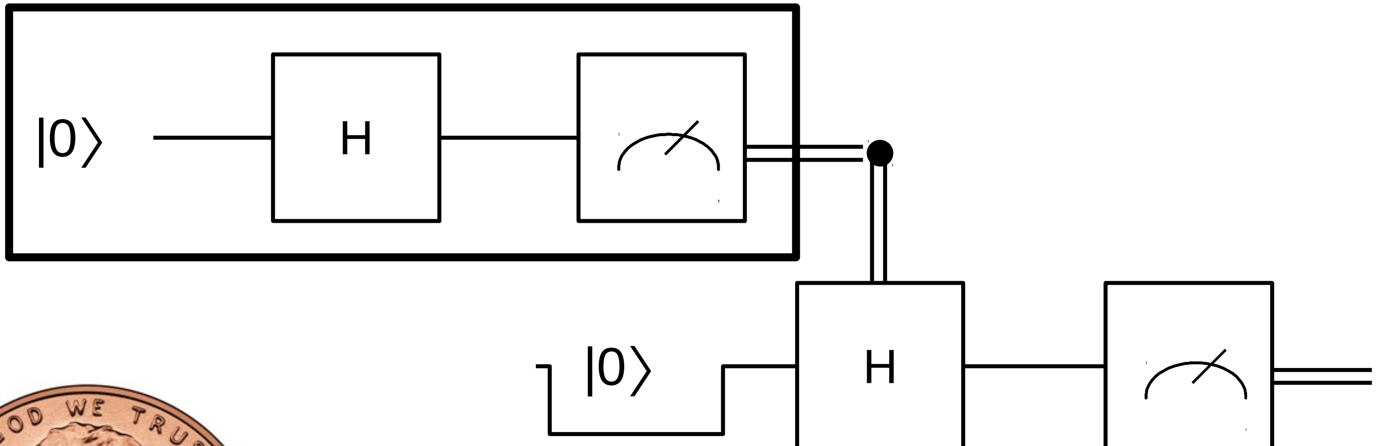
Features of Circuits

- gates applied to wires
- controlled circuits



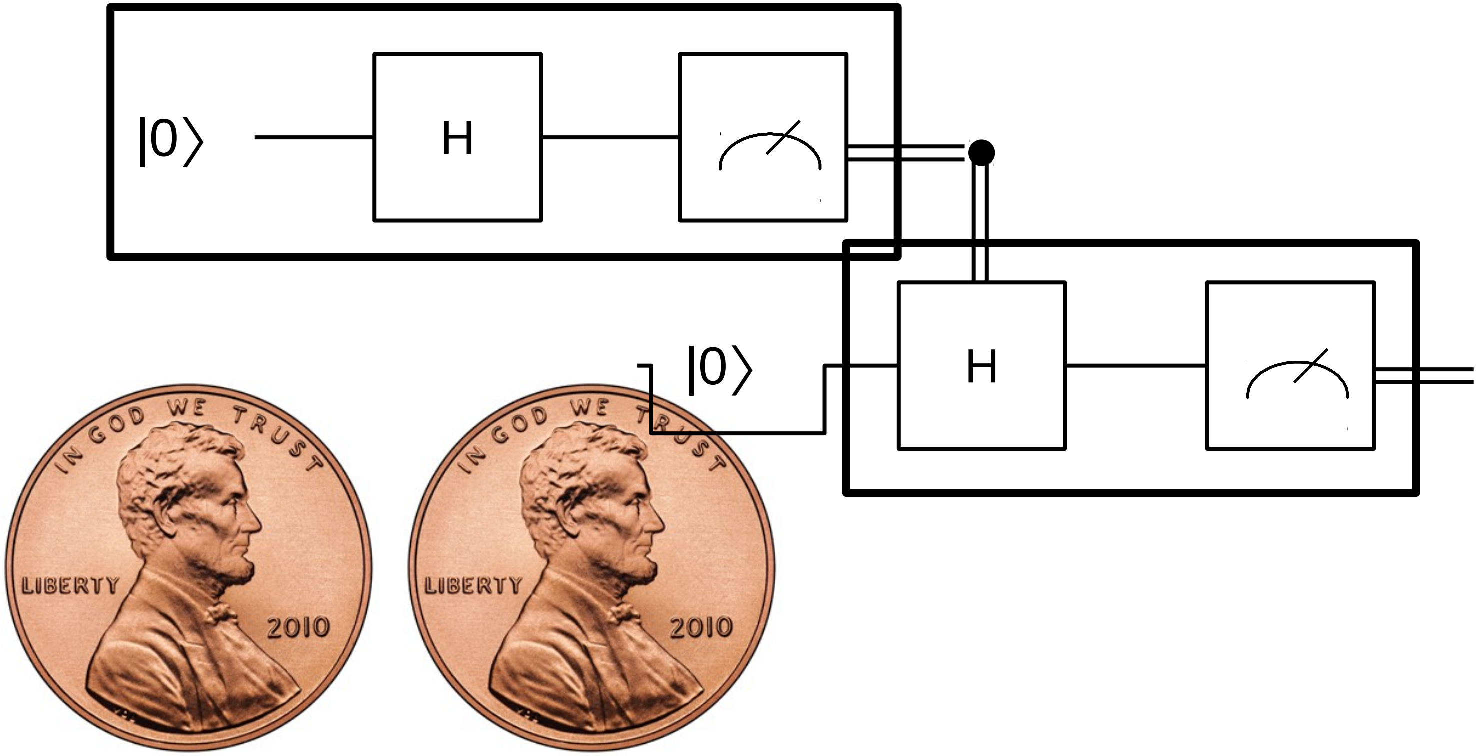
Features of Circuits

- gates applied to wires
- controlled circuits



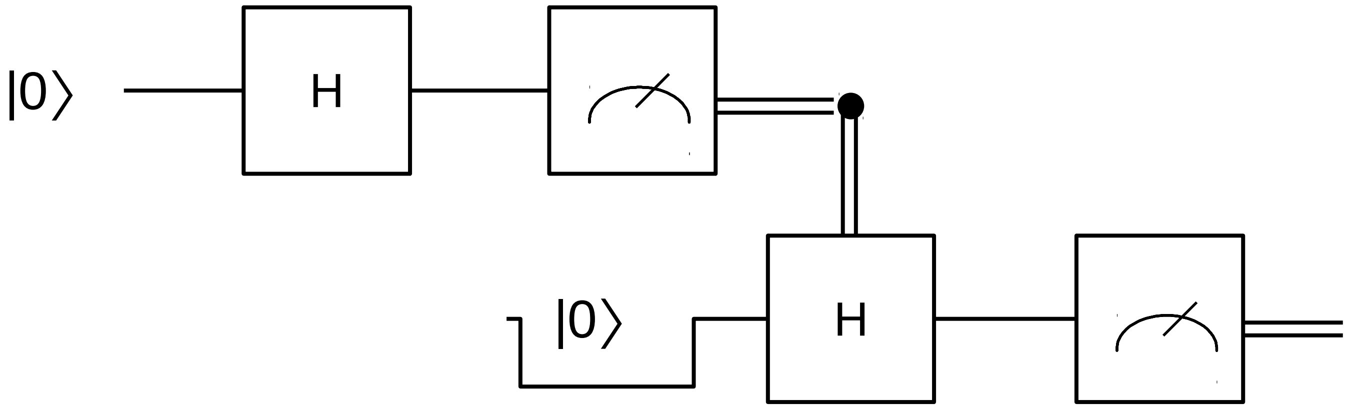
Features of Circuits

- gates applied to wires
- controlled circuits



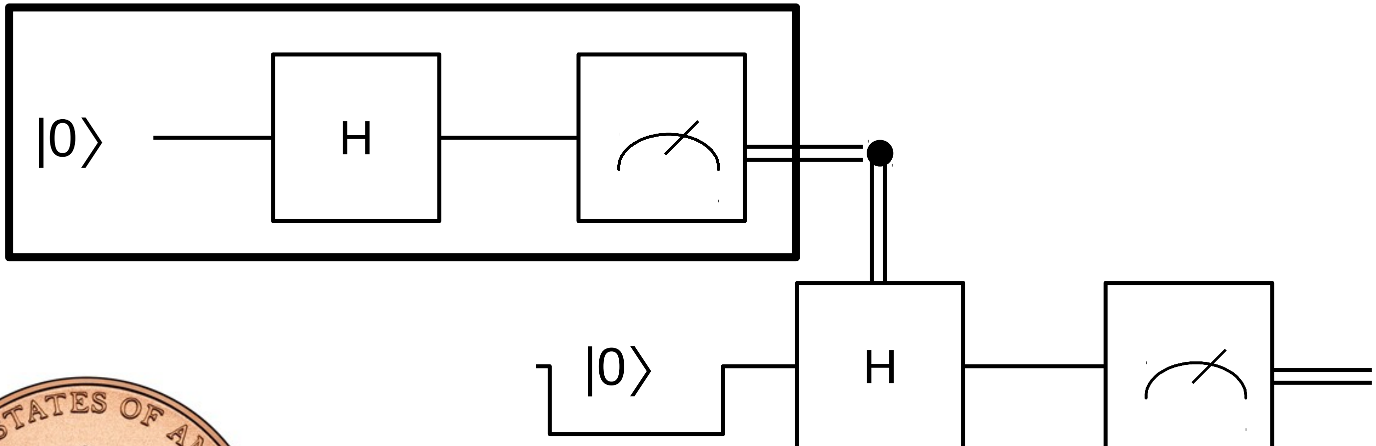
Features of Circuits

- gates applied to wires
- controlled circuits



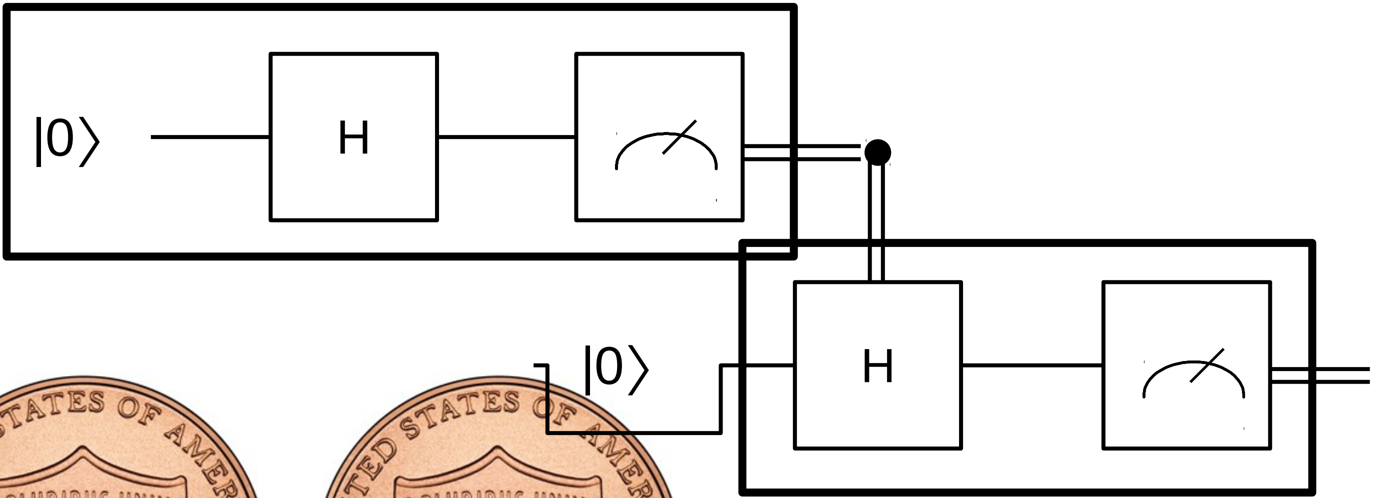
Features of Circuits

- gates applied to wires
- controlled circuits



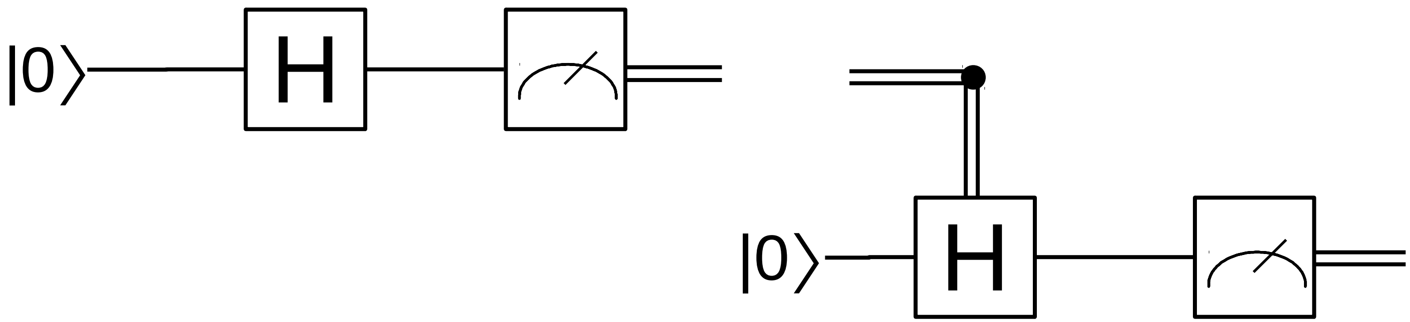
Features of Circuits

- gates applied to wires
- controlled circuits



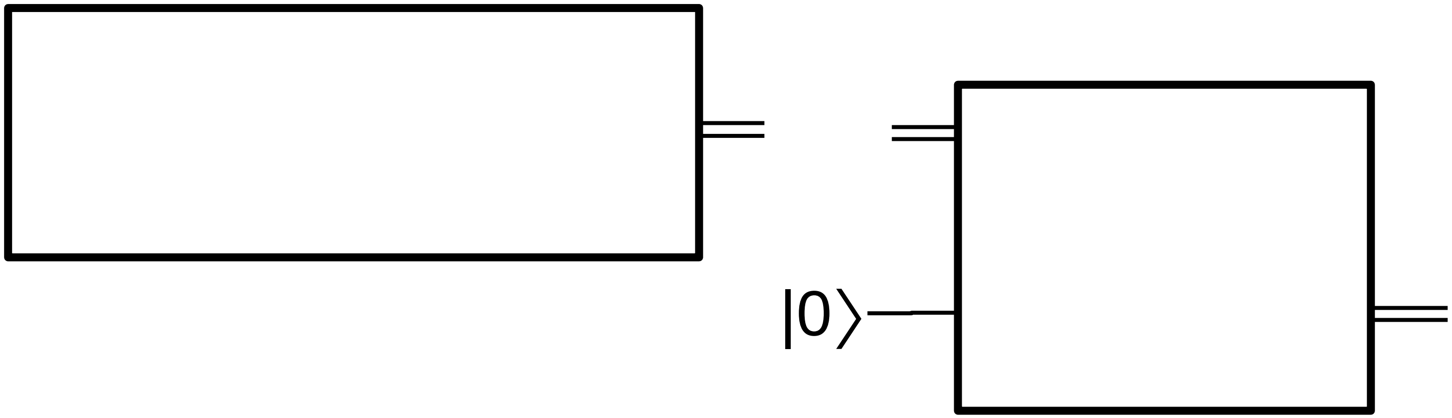
Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular



Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular

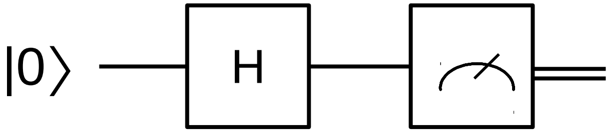


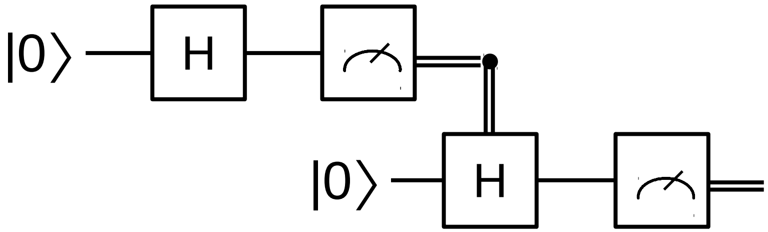
Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular
- families of circuits

Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular
- families of circuits



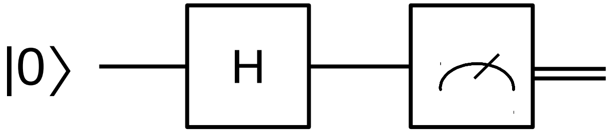


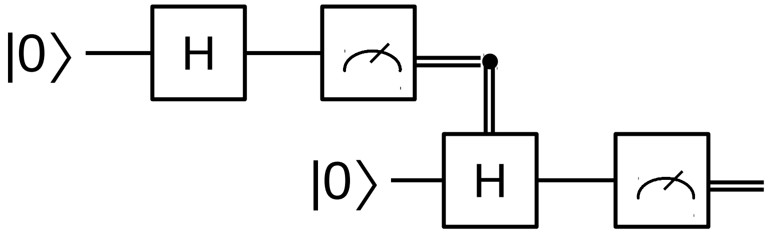
$$\left\{ \begin{array}{ll} 0 & \text{with probability } 1/2 \\ 1 & \text{with probability } 1/2 \end{array} \right\}$$

$$\left\{ \begin{array}{ll} 0 & \text{with probability } 3/4 \\ 1 & \text{with probability } 1/4 \end{array} \right\}$$

Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular
- families of circuits



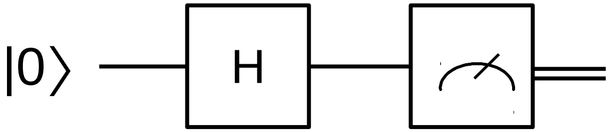


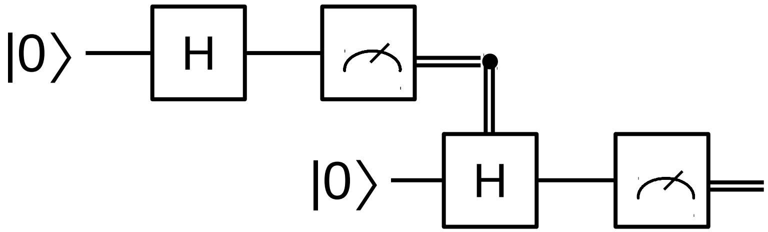
$$\left\{\begin{array}{ll} 0 & \text{with probability } 1/2 \\ 1 & \text{with probability } 1/2 \end{array}\right\}$$

$$\left\{\begin{array}{ll} 0 & \text{with probability } 3/4 \\ 1 & \text{with probability } 1/4 \end{array}\right\}$$

Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular
- families of circuits





$$\left\{ \begin{array}{ll} 0 & \text{with probability } 1/2 \\ 1 & \text{with probability } 1/2 \end{array} \right\}$$

$$\left\{ \begin{array}{ll} 0 & \text{with probability } 3/4 \\ 1 & \text{with probability } 1/4 \end{array} \right\}$$

Features of Circuits

- gates applied to wires
- controlled circuits
- compositional/modular
- families of circuits
- semantics—mathematical meaning

$$\left[\!\left| \texttt{flip n} \right|\!\right]() = \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

Quantum Circuit Languages

- Low-level – limited types, composition, and/or circuit families
  - PyQuil – Python framework for constructing Rigetti Quil circuits
  - QisKit – Python framework for constructing IBM QASM circuits
  - ProjectQ – open source framework for converting between low-level circuit languages, shared benchmarking, etc

Quantum Circuit Languages

- High level – types, composition, circuit families
  - Q# – standalone Microsoft language with simulation, optimization, debuggers
  - Quipper – functional language for circuit families embedded in Haskell
  - **QWIRE** – Strong type system with **semantics** embedded in Coq **proof assistant**

Quantum Circuit Languages

Quantum Circuit Languages

- Semantics??
  - Academic languages – mostly not implemented
  - Exception: Proto-Quipper (Selinger et al work in progress)

Quantum Circuit Languages

- Semantics??
  - Academic languages – mostly not implemented
  - Exception: Proto-Quipper (Selinger et al work in progress)
  - **QWIRE** – **High-level** circuit language with **semantics** embedded in Coq **proof assistant**

What could go wrong?

What could go wrong?

$$\left[\!\left[\ \texttt{flip n}\ \right]\!\right]() \neq \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

What could go wrong?

- Algorithm is wrong → pen & paper proof

$$\left[\!\left| \texttt{flip n} \right|\!\right] () \neq \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

What could go wrong?

- Algorithm is wrong → pen & paper proof
- Implementation doesn’t mach specification → ??

$$\left[\!\left|\ \texttt{flip n}\ \right|\!\right]() \neq \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

Implementation vs Specification

Implementation vs Specification

- Test it?

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware
  - Cannot “debug” in the classical way

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware
  - Cannot “debug” in the classical way
- Simulate it?

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware
  - Cannot "debug" in the classical way
- Simulate it?
  - Only feasible for small inputs, by definition

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware
  - Cannot “debug” in the classical way
- Simulate it?
  - Only feasible for small inputs, by definition
- Prove it?

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware
  - Cannot “debug” in the classical way
- Simulate it?
  - Only feasible for small inputs, by definition
- Prove it?
  - Need a semantics for your language

Implementation vs Specification

- Test it?
  - Probabilistic results require many tests
  - Limitations of current-gen quantum hardware
  - Cannot “debug” in the classical way
- Simulate it?
  - Only feasible for small inputs, by definition
- Prove it?
  - Need a semantics for your language
  - Pen & paper proofs miss implementation details

Formal Verification

Formal Verification

- Formal proof system based on small trusted core

Formal Verification

- Formal proof system based on small trusted core
- Formalize semantics of circuits inside proof system

Formal Verification

- Formal proof system based on small trusted core
- Formalize semantics of circuits inside proof system

```
C : Circuit          ρ : Density_Matrix
--------------------------------------
         ⟦C⟧ρ : Density_Matrix
```

Formal Verification

- Formal proof system based on small trusted core
- Formalize semantics of circuits inside proof system
- Prove statements about circuits

```
C : Circuit          ρ : Density_Matrix
--------------------------------------
        ⟦C⟧ρ : Density_Matrix
```

Formal Verification in Coq

- Formal proof system based on small trusted core
- Formalize semantics of circuits inside proof system
- Prove statements about circuits

QWIRE in Coq

https://github.com/inQWIRE/QWIRE

Trade-offs of Formal Verification

Trade-offs of Formal Verification

- Proofs directly connect code to spec
  - If code changes, proof changes

Trade-offs of Formal Verification

- Proofs directly connect code to spec
  - If code changes, proof changes
- Proofs are formal
  - Don’t let you skip any important details
  - Can use high-level reasoning techniques

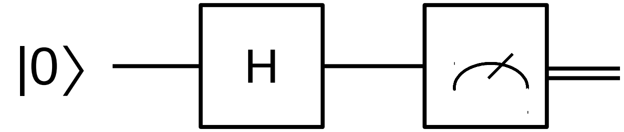
Trade-offs of Formal Verification

- Proofs directly connect code to spec
  - If code changes, proof changes
- Proofs are formal
  - Don't let you skip any important details
  - Can use high-level reasoning techniques
- It's a lot of work!

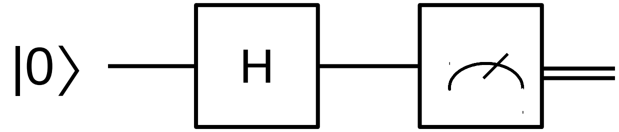
Trade-offs of Formal Verification

- Proofs directly connect code to spec
  - If code changes, proof changes
- Proofs are formal
  - Don’t let you skip any ~~important~~ details
  - Can use high-level reasoning techniques
- It’s a lot of work!

Formal Verification

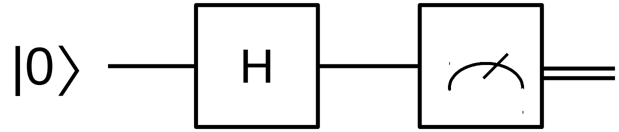


$$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$



Formal Verification

$$\texttt{Theorem fair\_toss : } [\![\texttt{coin\_flip}]\!]\ \texttt{I} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$



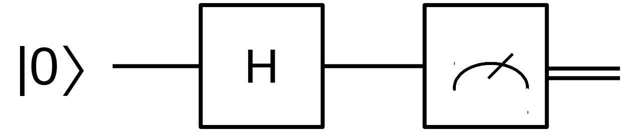
Formal Verification

`Theorem fair_toss : ⟦coin_flip⟧ I =` $\begin{pmatrix}\frac{1}{2} & 0 \\ 0 & \frac{1}{2}\end{pmatrix}$

`Proof.`

tactics

`Qed.`



Formal Verification

```
Theorem fair_toss : ⟦coin_flip⟧ I =
```

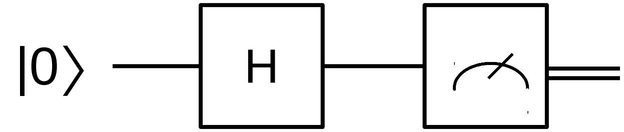
$$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

```
Proof.
```

tactics

```
Qed.
=============================
⟦ coin_flip ⟧ I = fair_coin
```

goal



Formal Verification

```
Theorem fair_toss : ⟦coin_flip⟧ I =
```

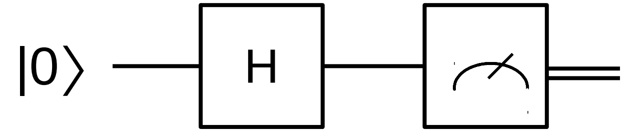
$$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

```
Proof.
  matrix_denote. (* unfold definitions *)


Qed.
```

* simplified for readability

```
  ============================
(I × I ⊗ I) × ((I ⊗ |0⟩⟨0| ⊗ I) × ((I ⊗ hadamard ⊗ I) ×
((I ⊗ |0⟩) × I × ((I ⊗ |0⟩)) †) × (I ⊗ hadamard ⊗ I) †)
× (I ⊗ |0⟩⟨0| ⊗ I) † .+ (I ⊗ |1⟩⟨1| ⊗ I) × ((I ⊗
hadamard ⊗ I) × ((I ⊗ |0⟩) × I × ((I ⊗ |0⟩)) †) × (I ⊗
hadamard ⊗ I) †) × (I ⊗ |1⟩⟨1| ⊗ I) †) × (I × I ⊗ I) †
= fair_coin
```



Formal Verification

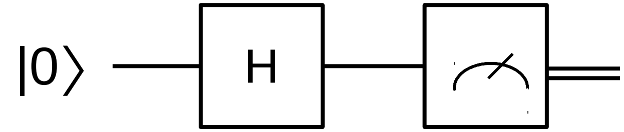
```
Theorem fair_toss : ⟦coin_flip⟧ I =
```

$$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

```
Proof.
  matrix_denote. (* unfold definitions *)
  Msimpl. (* simplify using known identities *)

Qed.

=======================
   |0⟩⟨0| × (hadamard × |0⟩⟨0| × hadamard) × |0⟩⟨0|
.+ |1⟩⟨1| × (hadamard × |0⟩⟨0| × hadamard) × |1⟩⟨1|
= fair_coin
```



Formal Verification

```
Theorem fair_toss : ⟦coin_flip⟧ I =
```

$$\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}$$

```
Proof.
  matrix_denote. (* unfold definitions *)
  Msimpl. (* simplify using known identities *)
  solve_matrix. (* brute force *)
Qed.

============================
No more subgoals.
```

Rule #1: Only rely on brute force for small cases

```
Fixpoint flip (n : Nat) :=
  match n with

  | 0    ⇒

  | S n' ⇒

  end.
```

$$\left\llbracket \left\lfloor \texttt{flip n} \right\rfloor \right\rrbracket () = \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

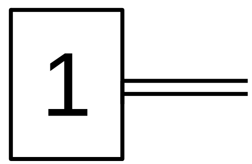
Rule #1: Only rely on brute force for small cases

```
Fixpoint flip (n : Nat) :=
  match n with

  | 0     ⇒

  | S n'  ⇒

  end.
```



$$\left[\!\left| \texttt{flip n} \right|\!\right] () = \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

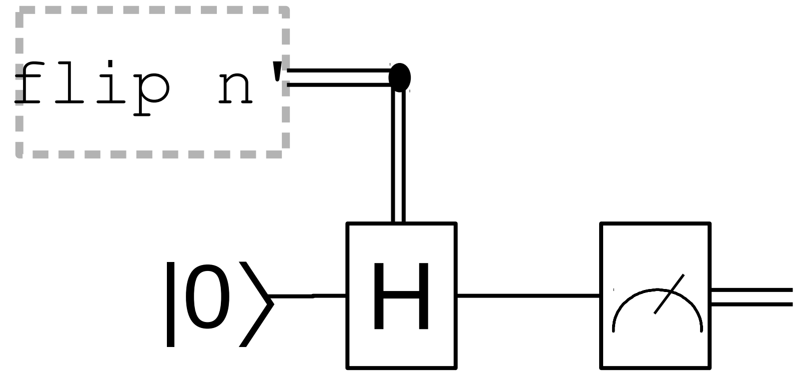
Rule #1: Only rely on brute force for small cases

```
Fixpoint flip (n : Nat) :=
  match n with

  | 0    ⇒

  | S n' ⇒

  end.
```



$$\left\llbracket \texttt{flip n} \right\rrbracket () = \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

Rule #2: Use induction to prove correctness of circuit families

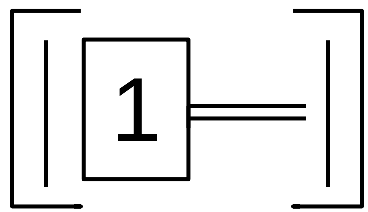
```
Lemma flip_corect : forall n,
```

$$\left[\!\left|\,\texttt{flip n}\,\right|\!\right]\big(\big) = \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

```
Proof.
  induction n.
```

Rule #2: Use induction to prove correctness of circuit families

```
Lemma flip_corect : forall n,
```

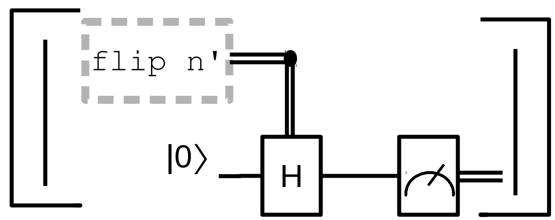


$$() = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$$

```
Proof.
  induction n.
  - (* n = 0 *) (* easy brute force *)
```

Rule #2: Use induction to prove correctness of circuit families

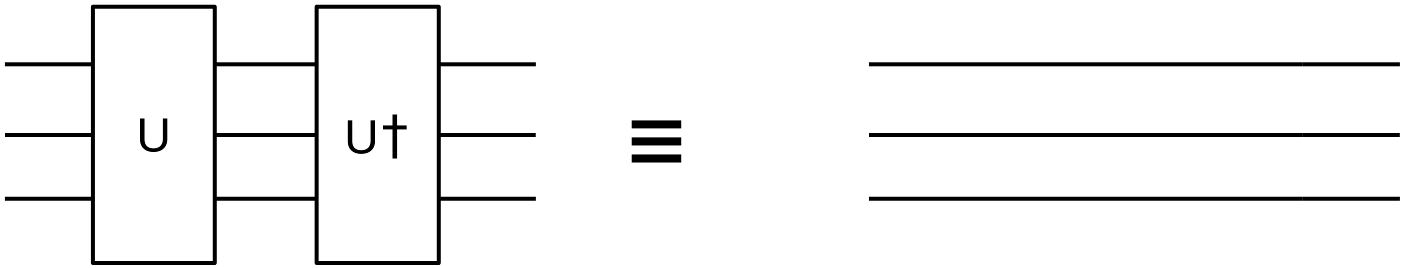
```
Lemma flip_corect : forall n,
```



$$() = \begin{pmatrix} \frac{2^n - 1}{2^n} & 0 \\ 0 & \frac{1}{2^n} \end{pmatrix}$$

```
Proof.
  induction n.
  - (* n = 0 *)     (* easy brute force *)
  - (* n = n'+1 *) (* use IH to complete proof *)
Qed.
```

Rule #3: Use equational reasoning




```
Lemma unitary_adjoint_id : ∀ n (U : Unitary n),
  U† ;; U ≡ id n.

(* C1 ≡ C2 := ∀ ρ, ⟦C1⟧ρ = ⟦C2⟧ρ *)
```

QWIRE: circuit language in Coq

- https://github.com/inQWIRE/QWIRE
- Verify particular circuits & protocols
  - teleportation, Deutsch’s algorithm
- Verify entire families of circuits
  - Inductively-defined circuits
- Verified compiler & compiler optimizations
  - Boolean expressions to quantum oracles

QWIRE: circuit language in Coq

- Minimal circuit language that has desired features
  - Not meant for production systems
  - Implemented in Coq—interactive theorem prover
- Strong type system prevents ill-formed circuits
- Denotational semantics
  - Formalized in Coq

Future work

Future work

- Current developer: Robert Rand at University of Maryland
  - Error rate analysis
  - Verified compiler passes

Future work

- Current developer: Robert Rand at University of Maryland
  - Error rate analysis
  - Verified compiler passes
- There are always trade-offs, but verification is often worthwhile!

|galois|

Thank you!

Formal Verification of Quantum Programs

Jennifer Paykin

https://github.com/inQWIRE/QWIRE

Pacific Northwest National Labs

October 26, 2018