A Verified Software Toolchain for Quantum Programming

Dissertation Defense

Kesha Hietala, May 16 2022

Techniques for classical program verification can be adapted to the quantum setting, allowing for the development of high-assurance quantum software, without sacrificing performance or programmability.

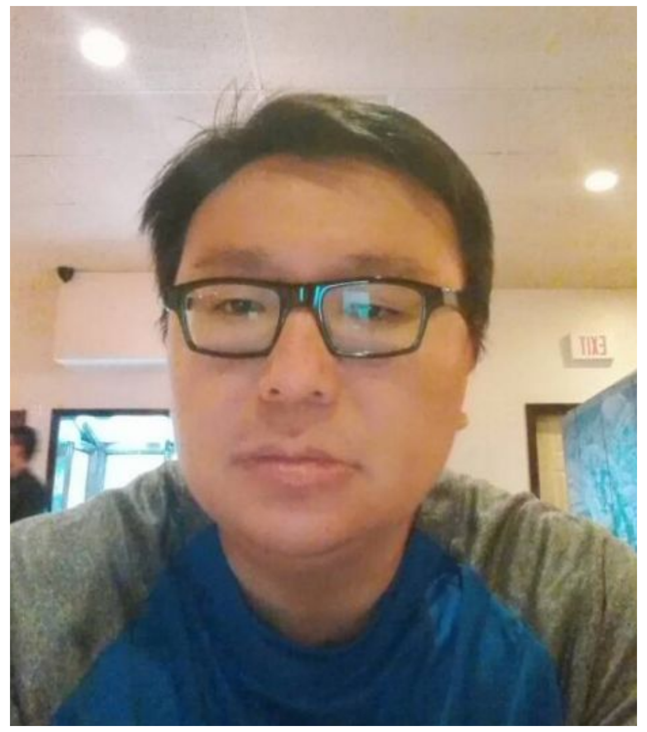
Acknowledgements

- Content based on:
  - **Kesha Hietala**, Robert Rand, Shih-Han Hung, Xiaodi Wu, Michael Hicks. *A Verified Optimizer for Quantum Circuits*. POPL 2021.
  - **Kesha Hietala**, Robert Rand, Shih-Han Hung, Liyi Li, Michael Hicks. *Proving Quantum Programs Correct*. ITP 2021.
  - Liyi Li, Finn Voichick, **Kesha Hietala**, Yuxiang Peng, Xiaodi Wu, Michael Hicks. *Verified Compilation of Quantum Oracles*. Draft.
  - Yuxiang Peng, **Kesha Hietala**, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, Xiaodi Wu. *A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm*. Draft.
  - Ongoing work with Sarah Marshall, Robert Rand, Kartik Singhal, Nik Swamy

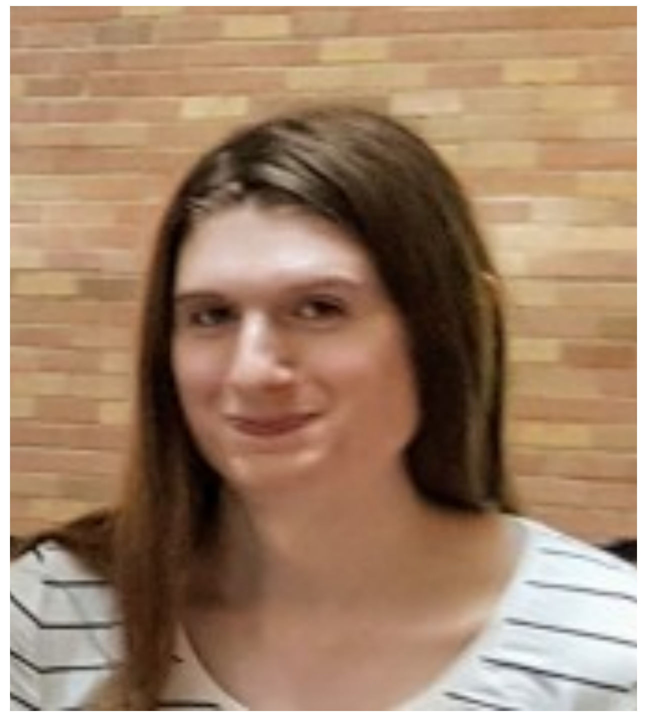
Acknowledgements

Mike Hicks

Shih-Han Hung



Liyi Li



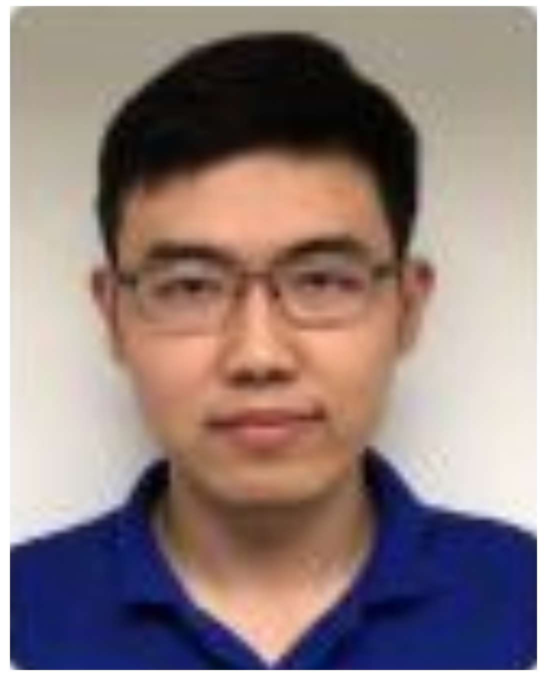
Sarah Marshall

Yuxiang Peng

Robert Rand

Kartik Singhal

Nik Swamy



Runzhou Tao

Finn Voichick

Xioadi Wu

Outline

Overview

Background

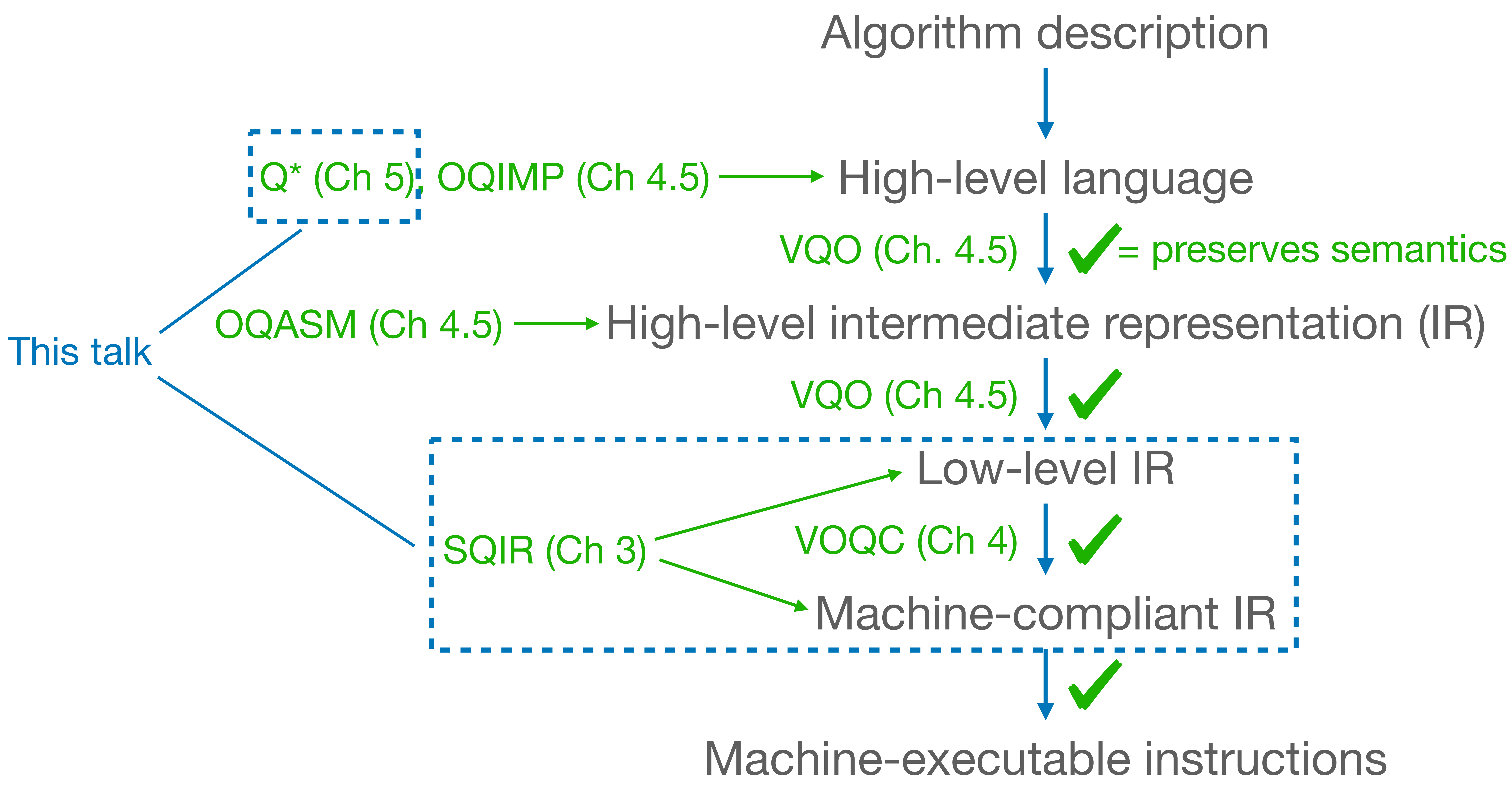
SQIR: A Small Quantum Language Supporting Verification

VOQC: A Verified Optimizer for Quantum Circuits

Q*: Formal Verification for a High-level Quantum Language

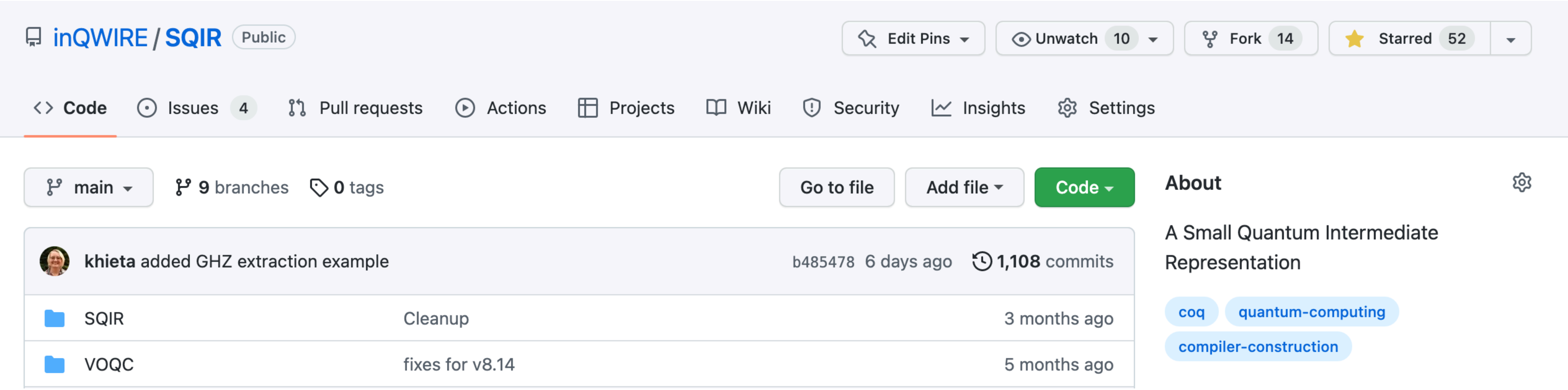
Summary

Verified Software Toolchain



Open Source Implementations

- SQIR/VOQC Coq impl. and proofs: github.com/inQWIRE/SQIR
- VOQC OCaml library: github.com/inQWIRE/mlvoqc
- VOQC Python bindings and tutorial: github.com/inQWIRE/pyvoqc
- VQO Coq impl. and proofs: github.com/inQWIRE/VQO



Outline

Overview

Background

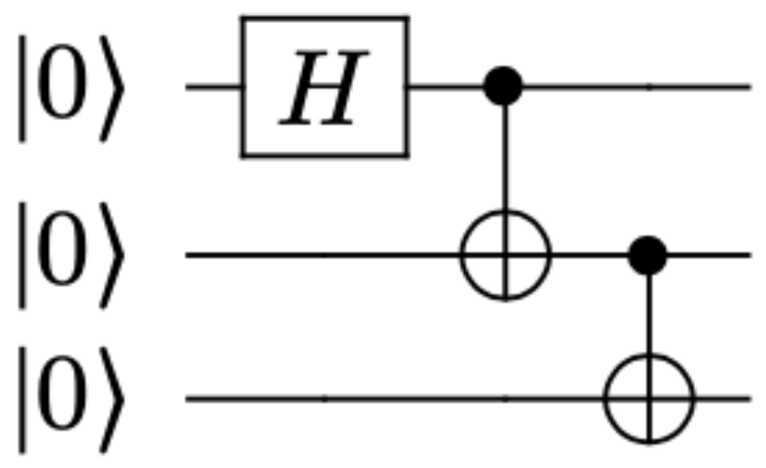
SQIR: A Small Quantum Language Supporting Verification

VOQC: A Verified Optimizer for Quantum Circuits

Q*: Formal Verification for a High-level Quantum Language

Summary

Quantum Programming



Circuit

```
H 0
CNOT 0 1
CNOT 1 2
```

QASM

```
def ghz_state(qubits):
  program = Program()
  program += H(qubits[0])
  for q1,q2 in zip(qubits, qubits[1:]):
    program += CNOT(q1, q2)
  return program
```

PyQuil

- Quantum programs are often described using circuits
- “High-level” quantum programming languages are often libraries for constructing circuits

Quantum Program Semantics

- States are interpreted as vectors/matrices; applying an operation is left-multiplication by a matrix

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |00\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad H|0\rangle = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

n qubits = 2^n-length vector

- *Example*: show that the program on the previous slide produces the state $1/\sqrt{2}(|000\rangle + |111\rangle)$

$|000\rangle$ —`H 0`→ $1/\sqrt{2}(|0\rangle + |1\rangle)|00\rangle = 1/\sqrt{2}(|000\rangle + |100\rangle)$

—`CNOT 0 1`→ $1/\sqrt{2}(|000\rangle + |110\rangle)$

—`CNOT 1 2`→ $1/\sqrt{2}(|000\rangle + |111\rangle)$

Verified Quantum Programming

- *Formal verification* is the process of proving that a program matches its specification

  Stronger guarantees than testing — specifications hold over all inputs

- Why should we formally verify quantum programs?
  - Can't debug on a quantum machine
    - *Current quantum machines are noisy and resource-limited*
    - *"printf( ) debugging" affects the state*
    - *Unit testing is expensive — programs output a distribution of results*
  - Can't debug (simulate) on a classical machine
    - *Requires resources exponential in the number of qubits*

Related Work

- Will existing classical verification technology "just work"?

  No, quantum programs have a very different semantics

- Is existing work on quantum formal verification sufficient?

	Quantum logics (QHL, qRHL) CAV 2019, POPL 2019	QWIRE POPL 2017	QBRICKS ESOP 2021	ReVerC CAV 2017	Giallar PLDI 2022	**SQIR & VOQC** POPL & ITP 2021
General semantics for quantum programs	✓	✓	✓−			✓
Applied to interesting quantum programs	✓		✓			✓+
Used to verify the software toolchain				✓	✓	✓
Contains state-of-the-art optimizations						✓

Outline

Overview

Background

SQIR: A Small Quantum Language Supporting Verification

VOQC: A Verified Optimizer for Quantum Circuits

Q*: Formal Verification for a High-level Quantum Language

Summary

SQIR

- A *Small Quantum Intermediate Representation*

$$U \ ::= \ U_1;\ U_2 \mid G\ q \mid G\ q_1\ q_2 \qquad \text{unitary core}$$

$$P \ ::= \ \texttt{skip} \mid P_1;\ P_2 \mid U \mid \texttt{meas}\ q\ P_1\ P_2 \qquad \text{full language}$$

- Semantics are matrices, parameterized by number of qubits

$$[\![\texttt{H 0; CNOT 0 1; CNOT 1 2}]\!]_3 = (I \otimes CNOT) \times (CNOT \otimes I) \times (H \otimes I \otimes I)$$

SQIR program

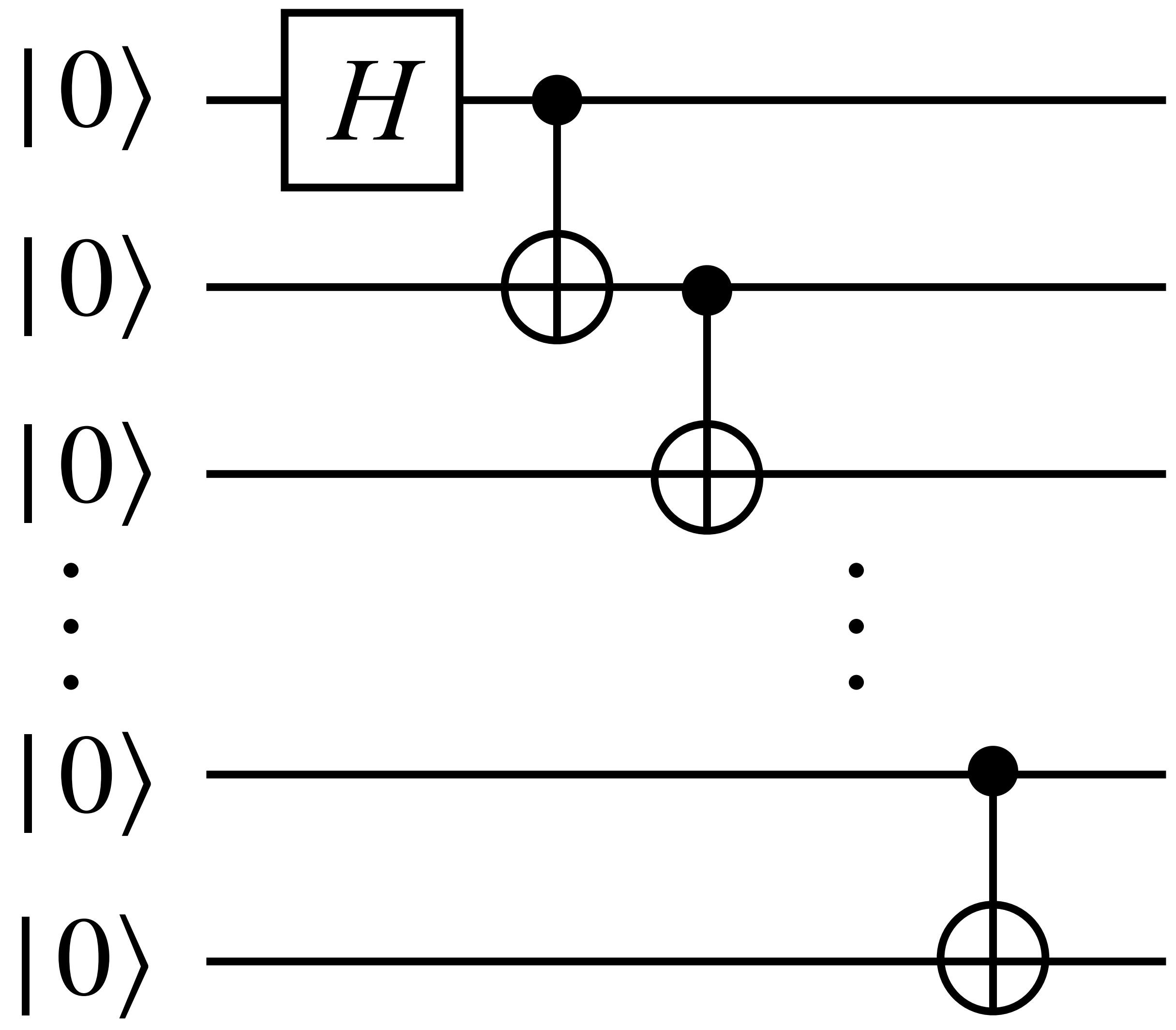
matrix expression

Denotation function

Three qubits, $2^3 \times 2^3$ matrix

Example: GHZ State Preparation

- The *n-qubit GHZ state*, $|GHZ^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$, is constructed by the following program




```
Fixpoint ghz (n : ℕ) : ucom base n :=
  match n with
  | 0 ⇒ SKIP
  | 1 ⇒ H 0
  | S n' ⇒ ghz n'; CNOT (n'-1) n'
  end.
```

- We prove that evaluating `ghz(n)` on $|0\rangle^{\otimes n}$ produces $|GHZ^n\rangle$

Proof "on paper"

- *Base*: Evaluating `ghz(1) = H 0` on $|0\rangle$ produces $|GHZ^1\rangle = 1/\sqrt{2}(|0\rangle + |1\rangle)$
- *Induction*: Assume the property holds for $k \geq 1$, i.e., evaluating `ghz(k)` on $|0\rangle^{\otimes k}$ produces $|GHZ^k\rangle = 1/\sqrt{2}(|0\rangle^{\otimes k} + |1\rangle^{\otimes k})$. Prove that the property holds for $k+1$.
  - Recall: `ghz(k+1) = ghz(k); CNOT (k-1) k`

$|0\rangle^{\otimes\, k+1}$ —`ghz(k)`→ $|GHZ^k\rangle|0\rangle$ —`CNOT (k-1) k`→ $1/\sqrt{2}\left(|0\rangle^{\otimes k}|0\rangle + |1\rangle^{\otimes k}|1\rangle\right)$

$|0\rangle^{\otimes\, k}|0\rangle$

$1/\sqrt{2}\left(|0\rangle^{\otimes k} + |1\rangle^{\otimes k}\right)|0\rangle$

$1/\sqrt{2}\left(|0\rangle^{\otimes k}|0\rangle + |1\rangle^{\otimes k}|0\rangle\right)$

$|GHZ^{k+1}\rangle$

Proof in Coq

- We write a correctness specification in Coq, and use tactics (e.g. “induction”) to construct a proof

```
Definition GHZ (n : ℕ) : Vector (2 ^ n) :=
    1/√2 * |0⟩^⊗n + 1/√2 * |1⟩^⊗n

Lemma ghz_correct : ∀ n : ℕ,
  n > 0 → ⟦ghz n⟧_n × |0⟩^⊗n = GHZ n.
Proof.
. . .
Qed.
```

- Coq checks that our proof is valid!

Design Highlights

- SQIR was conceived as a simplified version of QWIRE; we build on QWIRE's libraries for matrices and complex numbers
- We made several changes to simplify proof:
  - We reference qubits using concrete indices instead of variables

    CNOT (n - 1) n is easier to translate to a matrix than CNOT x y

  - We separate the unitary core from the full language w/ measurement

    Unitary matrices are simpler than functions over density matrices

See Ch 3.2 of my dissertation

Verified Quantum Algorithms

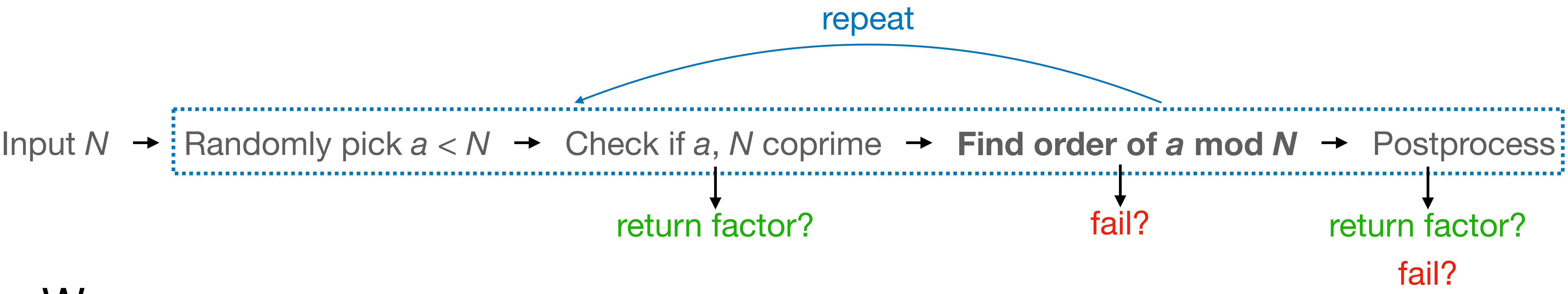
- Quantum teleportation Robert Rand
- Superdense coding
- GHZ state preparation Shih-Han Hung
- Deutsch-Jozsa algorithm Shih-Han Hung
- Simon's algorithm Liyi Li
- Quantum Fourier transform (QFT)
- Quantum phase estimation (QPE)
- Grover's algorithm
- Shor's factorization algorithm

Yuxiang Peng

Discussed in Ch 3.3 of my dissertation

Shor's Algorithm

- A probabilistic, hybrid quantum/classical algorithm for factoring numbers



- We prove:
  1. If the algorithm succeeds, it returns a factor
  2. Each iteration succeeds with probability at least $O((\log N)^{-4})$
  3. Each iteration uses $O((\log N)^3)$ quantum gates

Running Shor's Algorithm

- We *extract* our Coq definitions to OCaml code that generates SQIR circuits, which we then translate to OpenQASM 2.0

```
Fixpoint ghz (n : ℕ) : ucom base n :=
  match n with
  | 0 ⇒ SKIP
  | 1 ⇒ H 0
  | S n' ⇒ ghz n'; CNOT (n'-1) n'
  end.
```

→

```
let rec ghz n =
  if n=0 then coq_SKIP else
  if n=1 then coq_H 0 else
  Coq_useq (ghz (n−1), coq_CNOT (n−2) (n−1))
```

→

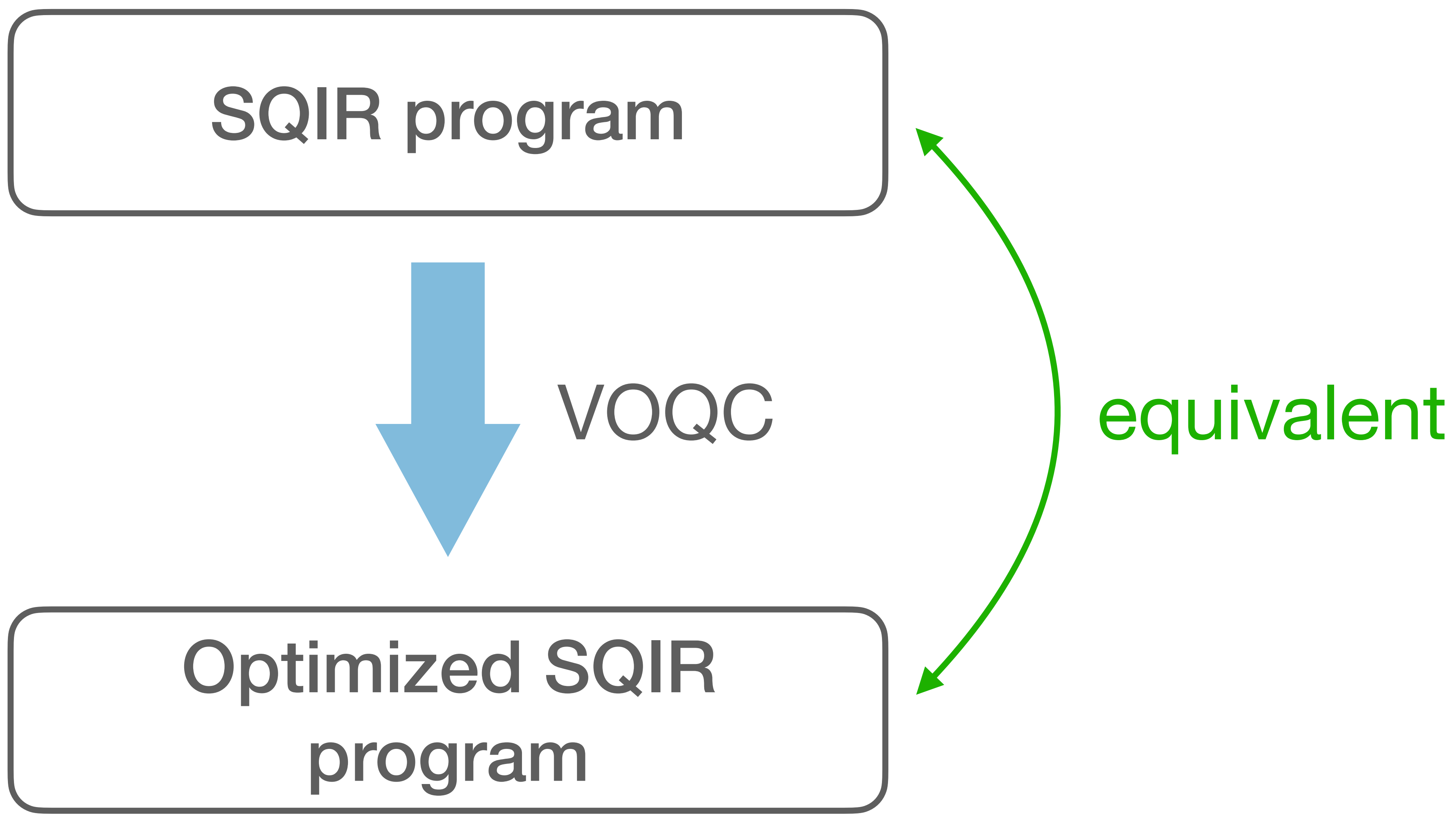
```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
h q[0];
cx q[0], q[1];
cx q[1], q[2];
```

- For Shor's, we also extract classical pre- and postprocessing to OCaml
- We replace execution on a quantum machine with a call to a classical simulator
  - Factoring 15 requires 35 qubits and ~22k gates
  - We've simulated factoring inputs up to 8 bits (< 256)

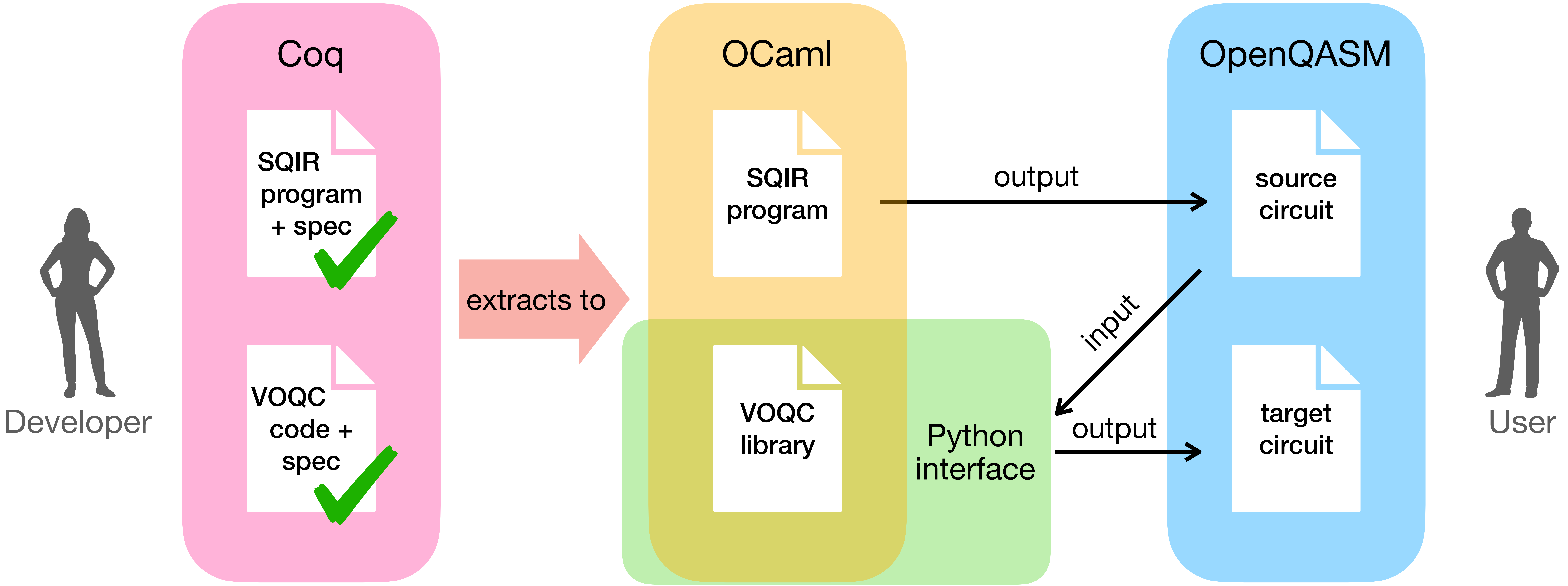
Outline

VOQC

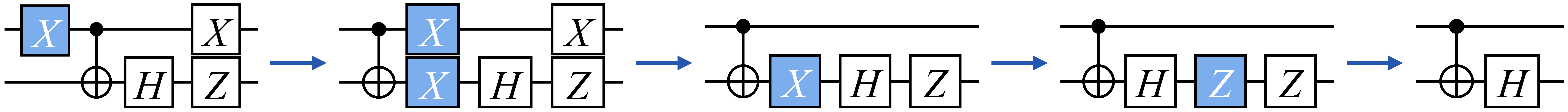
- A *Verified Optimizer for Quantum Circuits*
- We prove that transformations are semantics-preserving, i.e., they do not change the behavior of the program



VOQC Toolchain



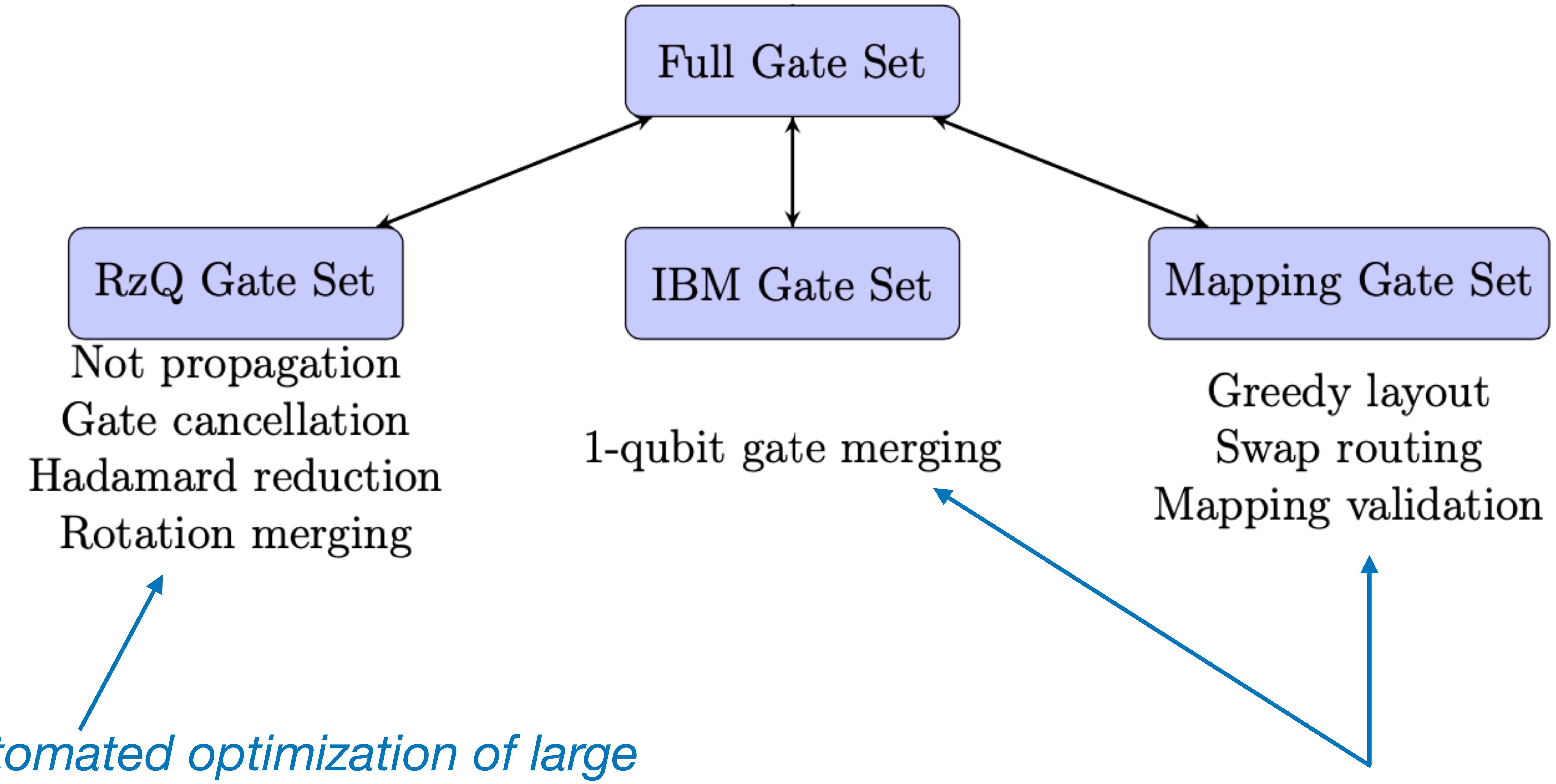
Example: Not Propagation



- At each step, the denotation of the program (i.e. unitary matrix) does not change
- We prove this via induction on the structure of the input program
  - ~30 lines to implement optimization
  - ~270 lines to prove soundness

Summary of VOQC Features

- VOQC supports **4** gates sets and **8** transformations



From Nam et al. *Automated optimization of large quantum circuits with continuous parameters*. npj Quantum Information.

From IBM's Qiskit compiler

Comparison w/ Other Optimizers

✓ = implemented in VOQC
✓* = VOQC contains a similar optimization

qiskit-terra 0.19.1	
Optimize1qGatesDecomposition	✓
CommutativeCancellation	✓*
ConsolidateBlocks w/ UnitarySynthesis	
pytket 0.19.2	
RemoveRedundancies	✓
FullPeepholeOptimise	
pystaq 2.1	
simplify	✓
rotation_fold	✓*
cnot_resynth	
pyzx 0.7.0	
full_optimize	✓*
full_reduce	

qiskit.org

Sivarajah et al. *tket: A retargetable compiler for NISQ devices*. Quantum Science & Technology.

Amy and Gheorghiu. *staq — A full-stack quantum processing tool.* Quantum Science & Technology.

Kissinger and van de Wetering. *PyZX: Large scale automated diagrammatic reasoning*. EPTCS.

Performance

	Qiskit	t|ket⟩	VOQC
Total gate count	13.7%	17.1%	27.4%
Two-qubit gate count	2.3%	2.8%	10.9%

(i) Gate count reduction for IBM gate set

	Staq	PyZX	VOQC
Total gate count	18.2%	-22.6%	30.2%
Two-qubit gate count	0.6%	-50.7%	10.9%
T-gate count	41.5%	42.5%	37.6%

(ii) Gate count reduction for RzQ gate set

Qiskit	t|ket⟩	Staq	PyZX	VOQC
0.70s	0.13s	0.03s	25.98s	0.12s

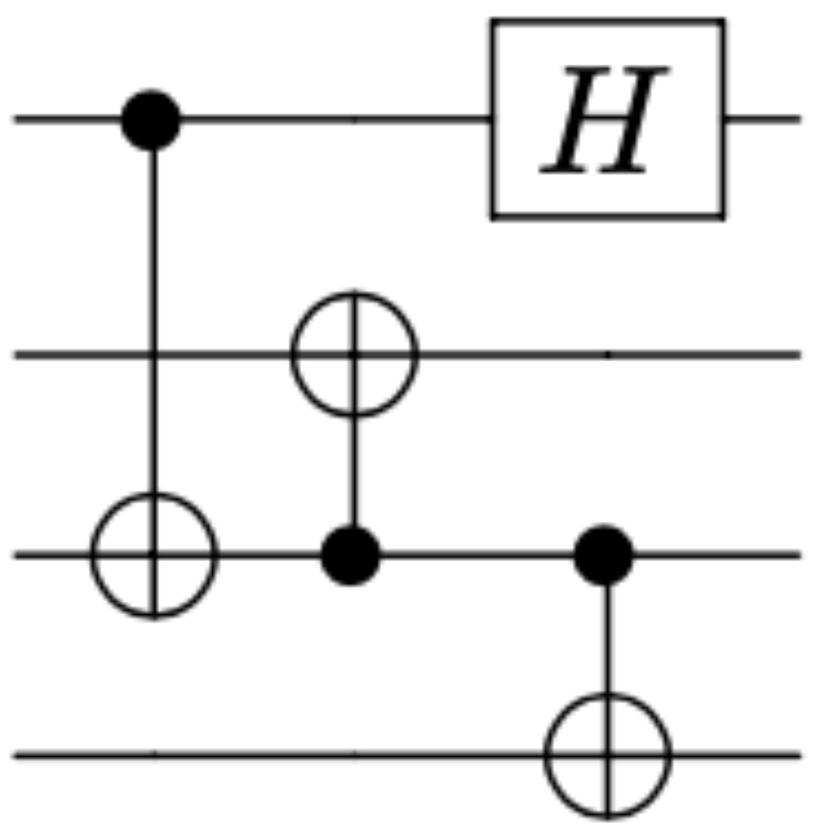
(iii) Running times

VOQC optimizes circuits better than existing optimizers, with comparable running time, and is also verified.

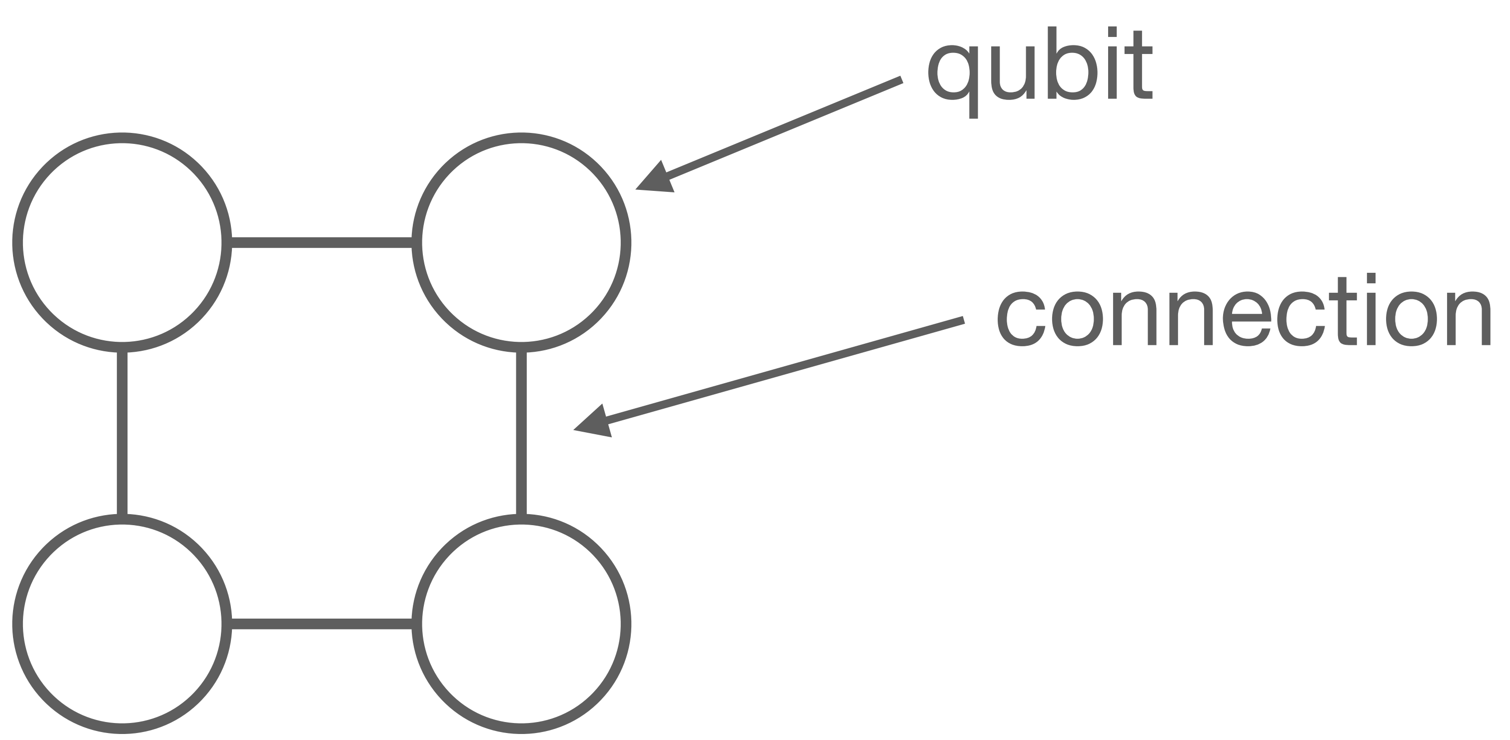
Circuit Mapping

- Given an input program & description of machine connectivity, mapping produces a program that meets connectivity constraints

E.g., how can we run the program on the left on the machine on the right?



4-qubit program

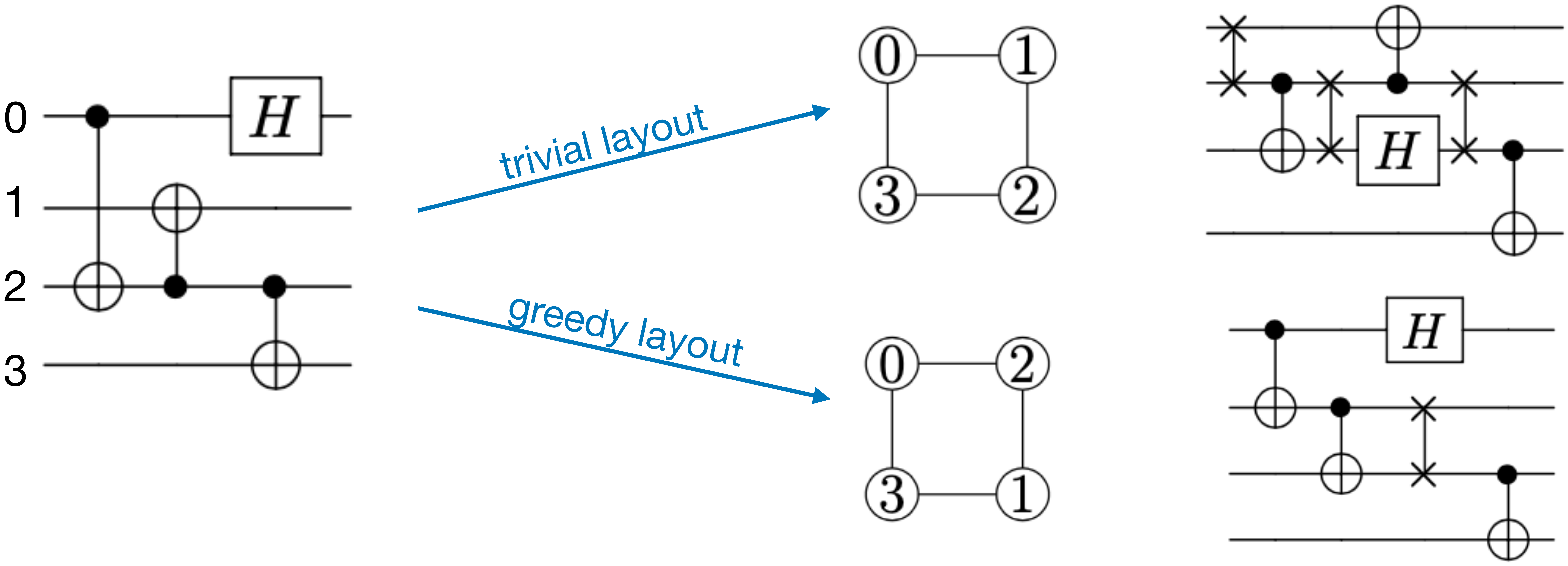


4-qubit machine

Circuit Mapping

- Given an input program & description of machine connectivity, mapping produces a program that meets connectivity constraints

E.g., how can we run the program on the left on the machine on the right?

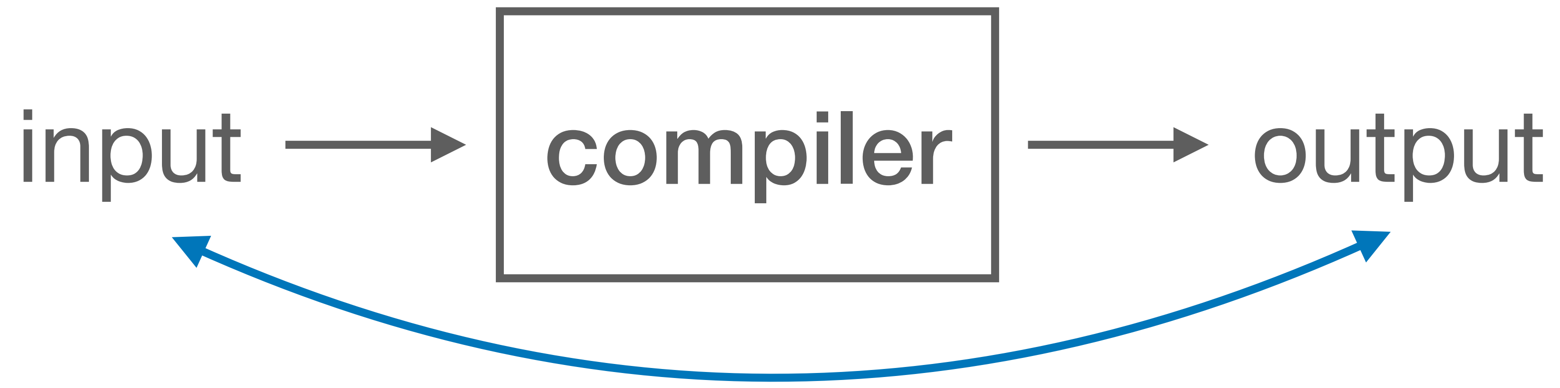


Verified Circuit Mapping

- For mapping, we prove that the output program is equivalent to the original, up to a permutation of qubits

$$\llbracket U_1 \rrbracket_d = P_1 \times \llbracket U_2 \rrbracket_d \times P_2$$

- To support more complex algorithms, we provide translation validation



translation validation = equivalence check

- We prove that if our translation validation succeeds, then the two programs are mathematically equivalent

Outline

Q# Programming Language

- A recent high-level quantum programming language from Microsoft docs.microsoft.com/en-us/azure/quantum/user-guide/
- Looks like a classical imperative programming language, but has some quantum-specific features

```
operation PrepareGhz (qs : Qubit[]) : Unit {
    H (qs[0]);
    ApplyCNOTChain(qs);
}
```

Incorrect Q# Programs, Allowed by Compiler

```
use q = Qubit();
CNOT(q, q);
```

(i) Violates no-cloning

```
operation InitQubit () : Qubit {
    use q = Qubit();
    return q;
}

operation ApplyX() : Unit {
    let q = InitQubit ();
    X(q);
}
```

allocates a fresh qubit

deallocates at the end of scope

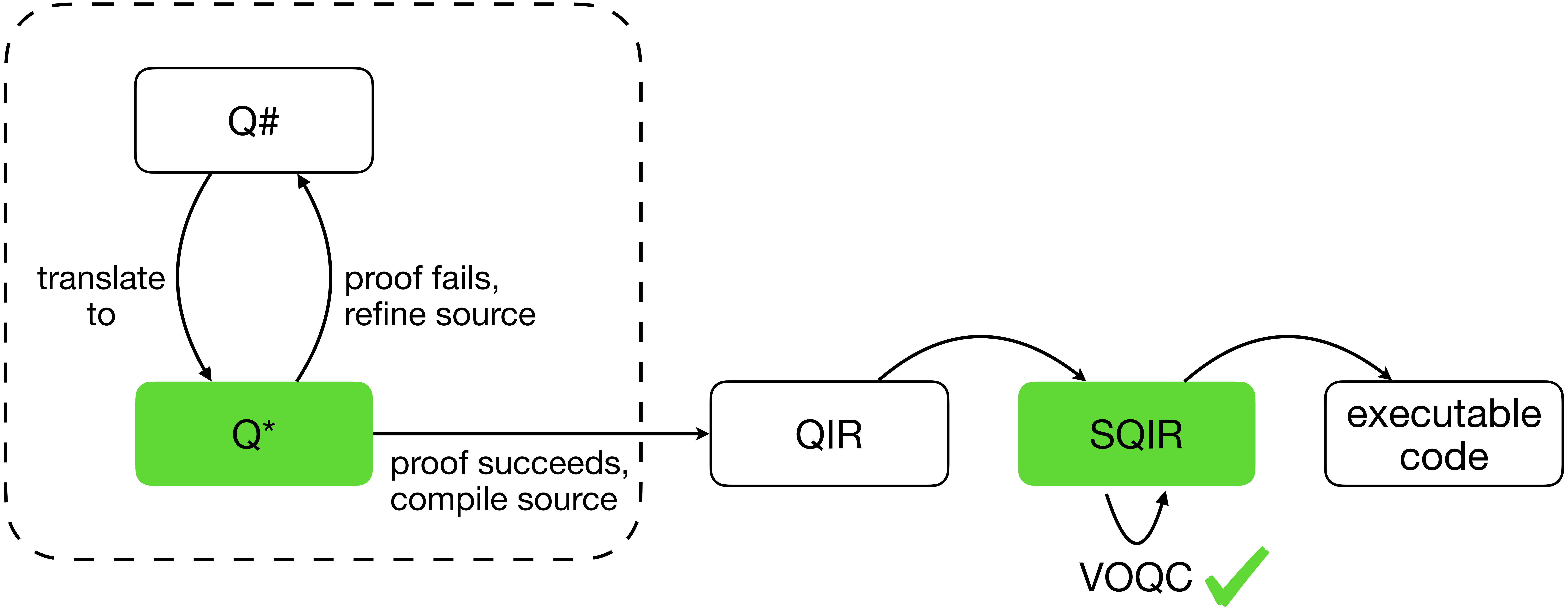
(ii) Reuses a discarded qubit

```
operation PrepareBell (q1 : Qubit)
        : Unit {
    use q2 = Qubit();
    H(q1);
    CNOT(q1, q2);
}
```

(iii) Discards an entangled qubit

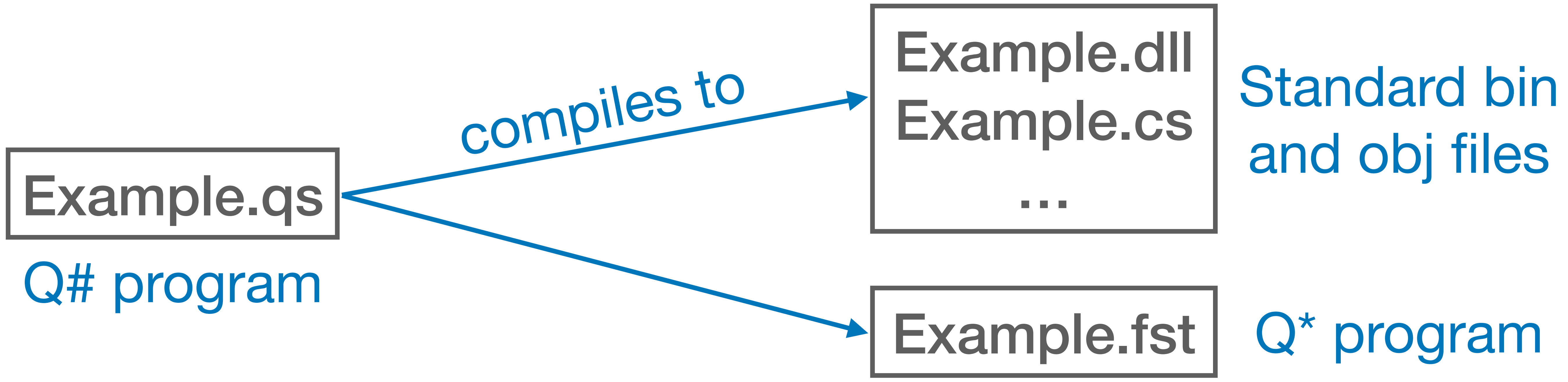
Q# + F* = Q*

F*: A proof-oriented programming language from Microsoft Research



Prototype Implementation

- We wrote a plugin for the Q# compiler that generates a Q* program



- We automatically generate simple specifications that the F* type checker will try to enforce

- We can also prove more general properties about the Q* program's semantics

Enforcing Linear Qubit Usage

- Linear qubit usage requires that all qubits used in a gate are *live* and *distinct*, and operations begin and end with the same live qubits

```
operation InitQubit () : Qubit {
    use q = Qubit();
    return q;
}

operation ApplyX() : Unit {
    let q = InitQubit ();
    X(q);
}
```

q is not live

```
use q = Qubit();
CNOT(q, q);
```

inputs are not distinct

- The type of InitQubit says that no new qubits were allocated
- The type of X says that the input must be live
- The type of CNOT says that the inputs must be live and distinct

Enforcing Discard Safety

- Qubits should be *unentangled* when they are discarded

```
operation PrepareBell (q1 : Qubit) : Unit {
    use q2 = Qubit();
    H(q1);
    CNOT(q1, q2);
}
```

q2 is entangled with q1

- A useful tool for reasoning about entanglement in quantum programs is *separation logic* from classical program analysis[2]

- We are working on building a quantum separation logic to enforce discard safety on top of Steel, a concurrent separation logic embedded in F*

[2]As proposed by Zhou et al. at LICS 2021 and Le et al. at POPL 2022

Outline

Overview

Background

SQIR: A Small Quantum Language Supporting Verification

VOQC: A Verified Optimizer for Quantum Circuits

Q*: Formal Verification for a High-level Quantum Language

Summary

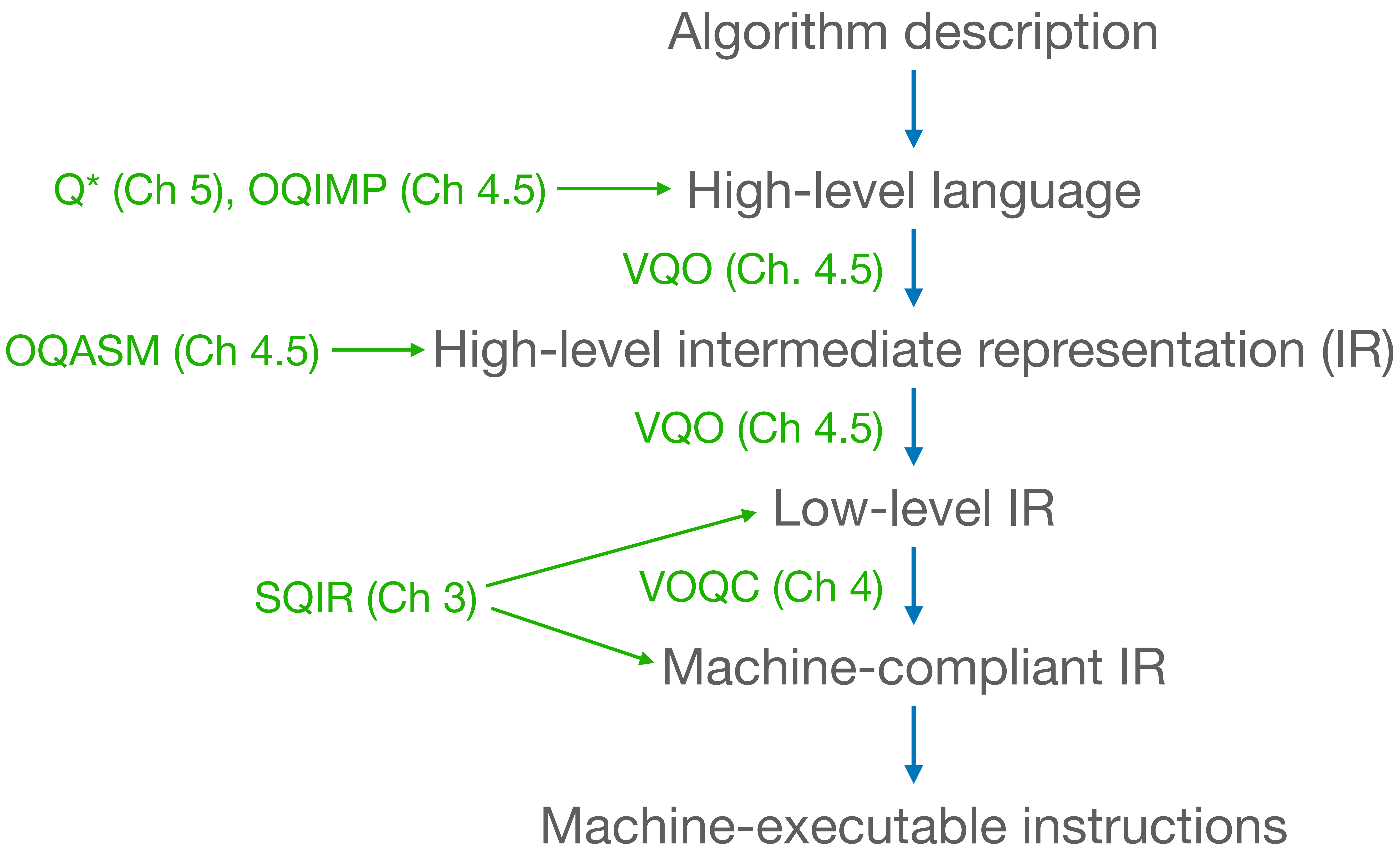
Techniques for classical program verification can be adapted to the quantum setting, allowing for the development of high-assurance quantum software, without sacrificing performance or programmability.

Contributions

To demonstrate this thesis, we presented:

- **SQIR,** which we have used to verify implementations of key quantum algorithms
- **VOQC**, a verified optimizer with performance on par with unverified tools Qiskit, tket, PyZX, and Staq
- **Q***, an initial effort to provide verification for the high-level language Q#

Future Directions



Future Directions

Algorithm description

↓ ? ← Better high-level languages to encode quantum algorithms

High-level language

↓ ? ← Compilers from high-level languages to circuits

High-level intermediate representation (IR)

↓ ? ← Compilers from high-level languages to circuits

Low-level IR

↓ ✓

Machine-compliant IR

↓ ? ← Compiler to machine-executable instructions (i.e., analog pulses)

Machine-executable instructions

Techniques for classical program verification can be adapted to the quantum setting, allowing for the development of high-assurance quantum software, without sacrificing performance or programmability.