A Verified Optimizer for Quantum Circuits

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, Michael Hicks

&

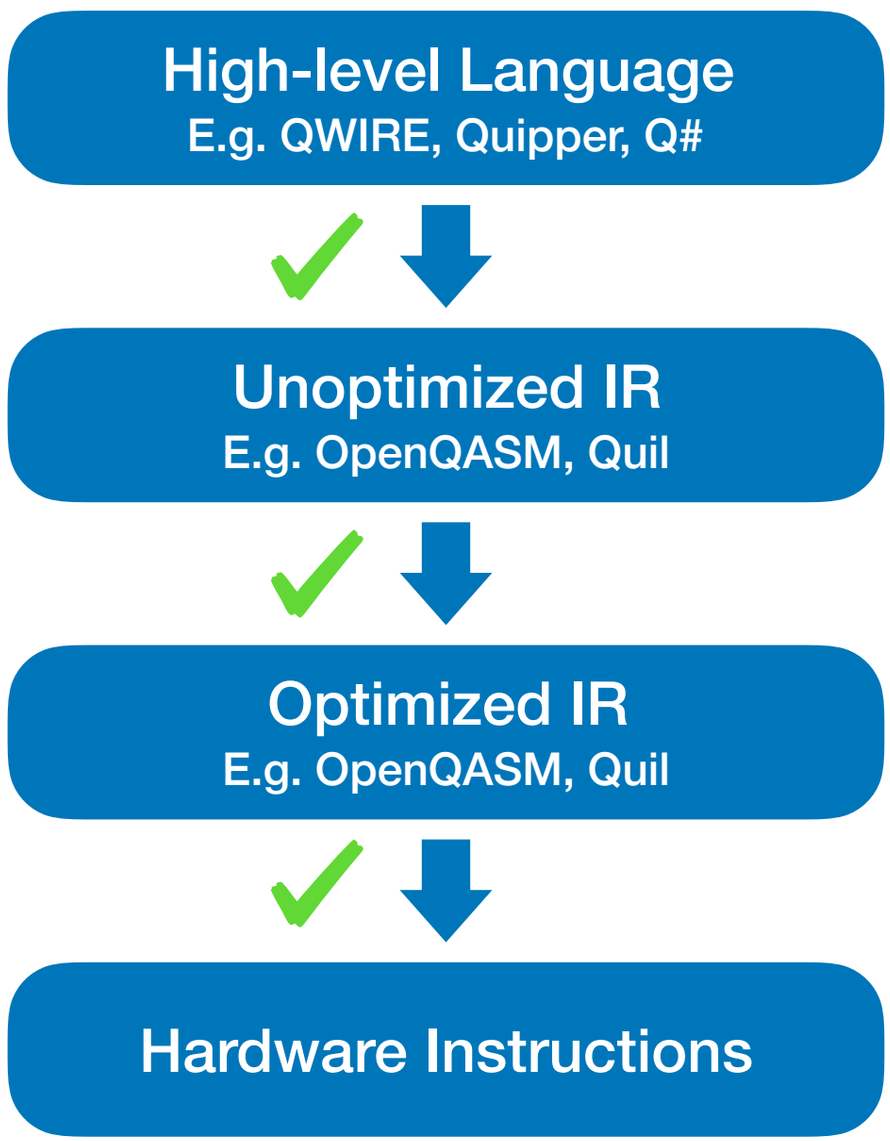
Verified translation between low-level quantum languages

Kartik Singhal, Robert Rand, Michael Hicks

PLanQC 2020

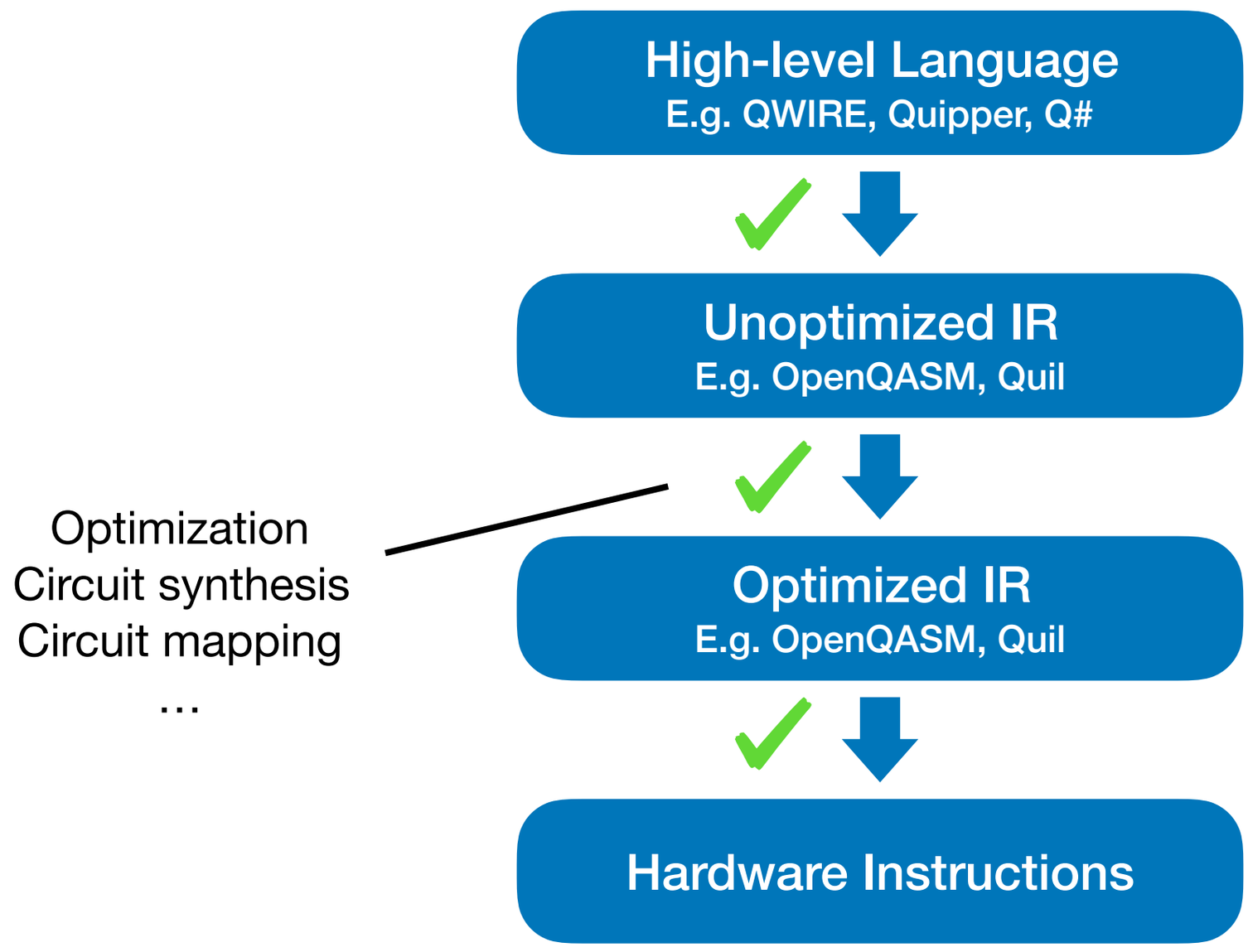
Verified Compiler Stack

- End goal: *verified compiler stack* for quantum programs



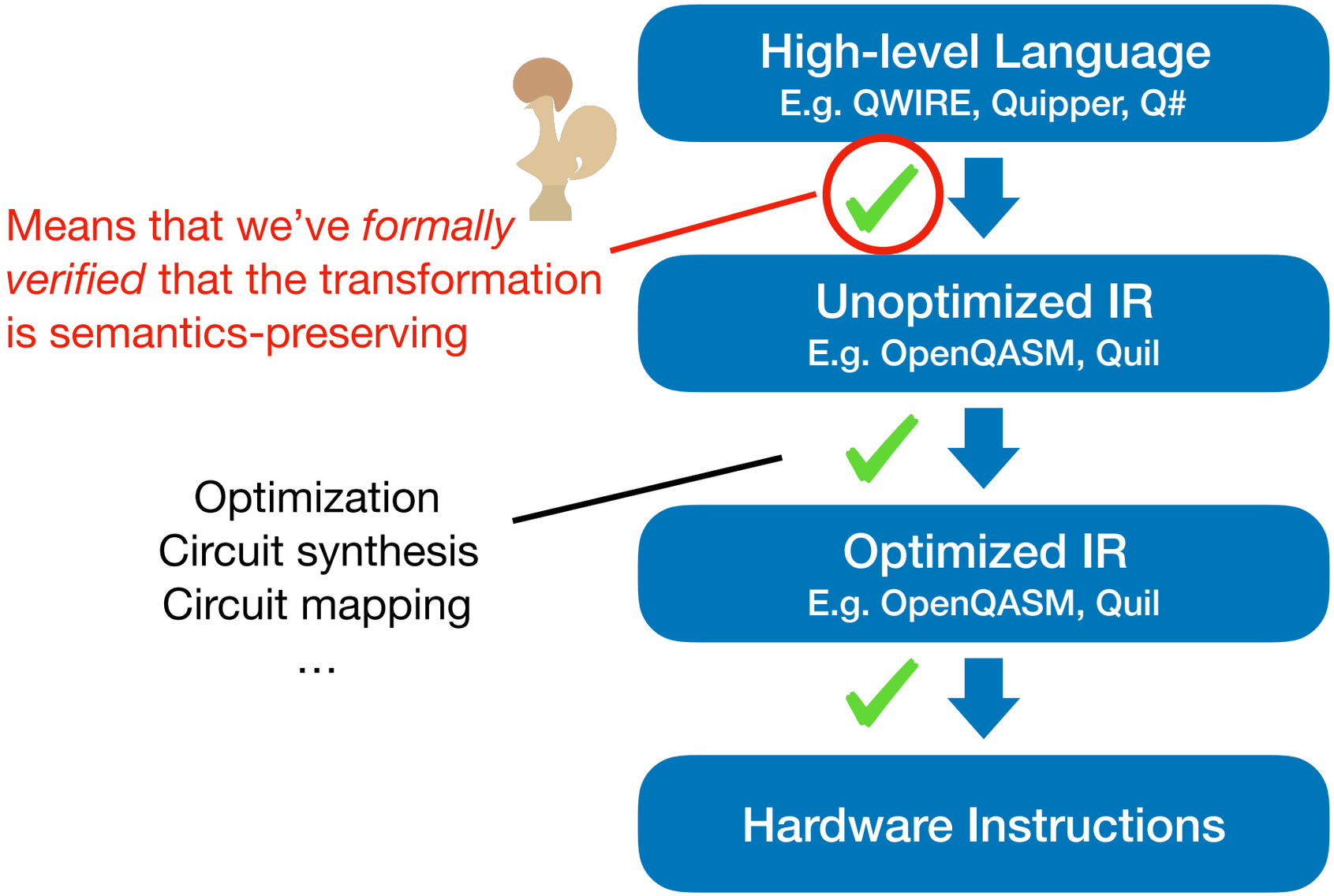
Verified Compiler Stack

- End goal: *verified compiler stack* for quantum programs



Verified Compiler Stack

- End goal: *verified compiler stack* for quantum programs



Verified Compiler Stack

- We present **VOQC**, our Verified Optimizer for Quantum Circuits, which is built on top of **SQIR**, our Small Quantum Intermediate Representation

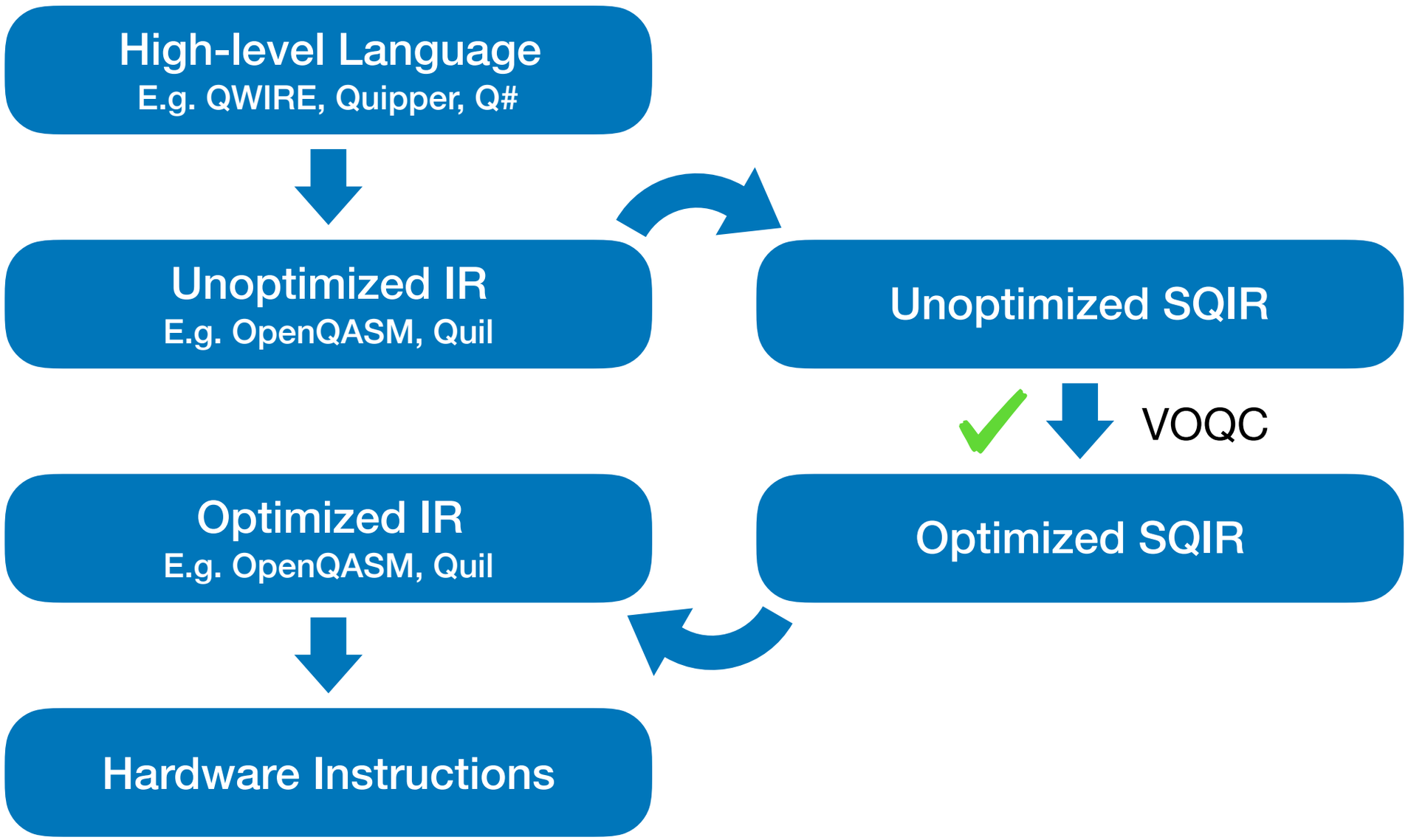
- Implemented in 8000 lines of Coq code, with 1500 for core SQIR and the rest for program transformations

- 400 lines of standalone OCaml code for parsing and translating OpenQASM

- We extract VOQC to OCaml and compile it to a binary, so using VOQC doesn’t require knowledge of Coq or OCaml

Verified Compiler Stack

- End goal: *verified compiler stack* for quantum programs



SQIR

- Syntax

$$U \;:=\; U_1;\; U_2 \;\big|\; G\; q \;\big|\; G\; q_1\; q_2$$

$$P \;:=\; \texttt{skip} \;\big|\; P_1;\; P_2 \;\big|\; U \;\big|\; \texttt{meas}\; q\; P_1\; P_2$$

- Semantics assumes a **global register** of size *d*
  - A unitary program corresponds to a unitary matrix of size $2^d \times 2^d$
  - A non-unitary program corresponds to a function between density matrices of size $2^d \times 2^d$

SQIR

- Syntax

$$U \;:=\; U_1;\; U_2 \;\big|\; G\; q \;\big|\; G\; q_1\; q_2$$

$$P \;:=\; \texttt{skip} \;\big|\; P_1;\; P_2 \;\big|\; U \;\big|\; \texttt{meas}\; q\; P_1\; P_2$$

- Semantics assumes a **global register** of size *d*

Unitary semantics

$$[\![U_1;\; U_2]\!]_d = [\![U_2]\!]_d \times [\![U_1]\!]_d$$

$$[\![G_1\; q]\!]_d = \begin{cases} apply_1(G_1,\; q,\; d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}$$

$$[\![G_2\; q_1\; q_2]\!]_d = \begin{cases} apply_2(G_2,\; q_1,\; q_2,\; d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}$$

Non-unitary semantics

$$\{\!|\texttt{skip}|\!\}_d(\rho) = \rho$$

$$\{\!|P_1;\; P_2|\!\}_d(\rho) = (\{\!|P_2|\!\}_d \circ \{\!|P_1|\!\}_d)(\rho)$$

$$\{\!|U|\!\}_d(\rho) = [\![U]\!]_d \times \rho \times [\![U]\!]_d^\dagger$$

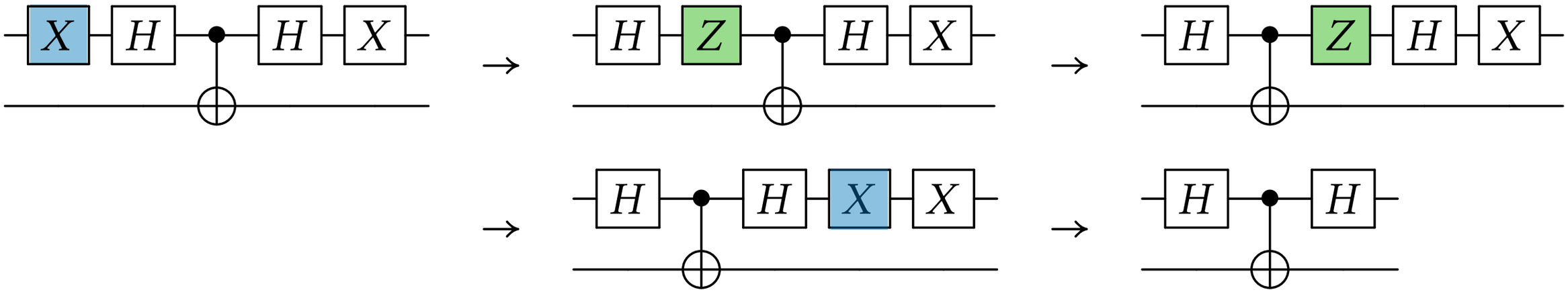
$$\{\!|\texttt{meas}\; q\; P_1\; P_2|\!\}_d(\rho) = \{\!|P_2|\!\}_d(|0\rangle_q\langle 0| \times \rho \times |0\rangle_q\langle 0|) + \{\!|P_1|\!\}_d(|1\rangle_q\langle 1| \times \rho \times |1\rangle_q\langle 1|)$$

VOQC Transformations

- Unitary optimizations - inspired by Nam et al.[1]
  - Gate propagation and cancellation
  - Rotation merging
- Non-unitary optimizations
  - Classical state propagation
  - Removing *z*-rotations before measurement
- Circuit mapping
  - Naive mapping for arbitrary connected graph

[1]Nam et al. *Automated Optimization of Large Quantum Circuits with Continuous Parameters*. 2018.

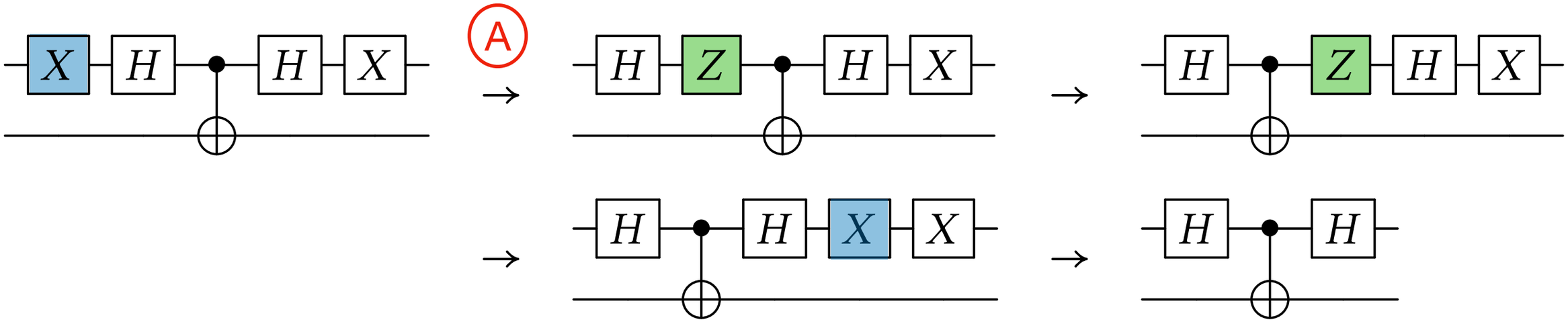
Example: X/Z Propagation



- Simplified code:

```
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```

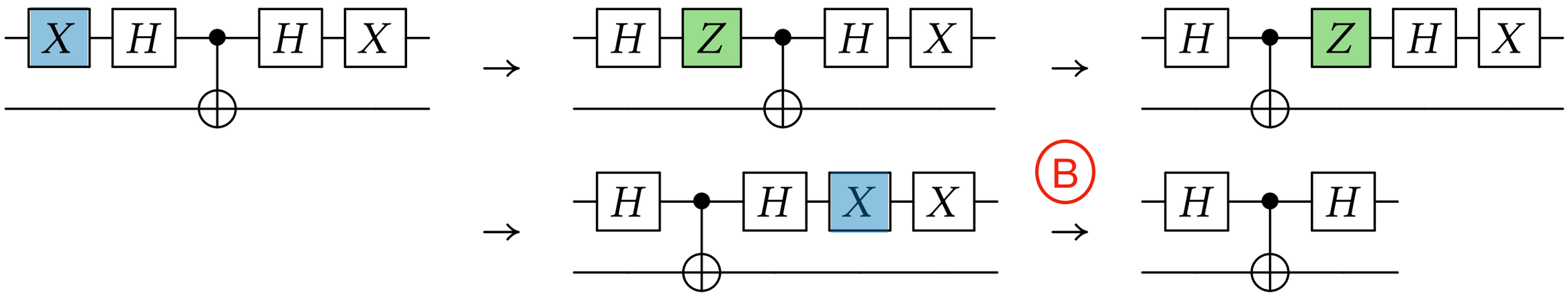
Example: X/Z Propagation



- Simplified code:

```
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t   (A)
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```

Example: X/Z Propagation



- Simplified code:

```
let propagate_X q lst = match lst with
  | []          → [X q]   (B)
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```

Proof Overview

- For a transformation T, we want to prove that
  $\forall P, [\![T(P)]\!] = [\![P]\!]$ (up to a global phase)
  - For complex optimizations, rather than proving this equality directly we may prove that $T(P)$ and P have the same output on every basis state

- For circuit mapping we also prove that for every P, $T(P)$ respects the provided connectivity constraints

- Proofs proceed by induction on P

Example: X/Z Propagation

- We will want to prove that for any instruction list lst, (propagate_X q lst) has the same denotation as (X q ; lst)
  - propagate_X q lst ≡ X q ; lst

- Proof proceeds by induction on lst

```
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```

Example: X/Z Propagation

- We will want to prove that for any instruction list lst, (propagate_X q lst) has the same denotation as (X q ; lst)
  - propagate_X q lst ≡ X q ; lst

- Proof proceeds by induction on lst

```
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t   → Rz† q ; propagate_X q t
  ...
```

propagate_X q [] → [X q] ≡ X q ; []

Example: X/Z Propagation

- We will want to prove that for any instruction list lst, (propagate_X q lst) has the same denotation as (X q ; lst)
  - propagate_X q lst ≡ X q ; lst

- Proof proceeds by induction on lst

```
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```

propagate_X q (X q :: t) → t ≡ X q ; X q ; t

Example: X/Z Propagation

- We will want to prove that for any instruction list lst, (propagate_X q lst) has the same denotation as (X q ; lst)
  - propagate_X q lst ≡ X q ; lst

- Proof proceeds by induction on lst

```
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```

propagate_X (Rz q :: t)
→ Rz† q ; propagate_X q t
≡ Rz† q ; (X q ; t)
≡ X q ; Rz q ; t

Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious
- E.g. $X\ n;\ CNOT\ m\ n \equiv CNOT\ m\ n;\ X\ n$

$$apply_1(X, n, d) \times apply_2(CNOT, m, n, d) = apply_2(CNOT, m, n, d) \times apply_1(X, n, d).$$

Verifying Matrix Equivalences

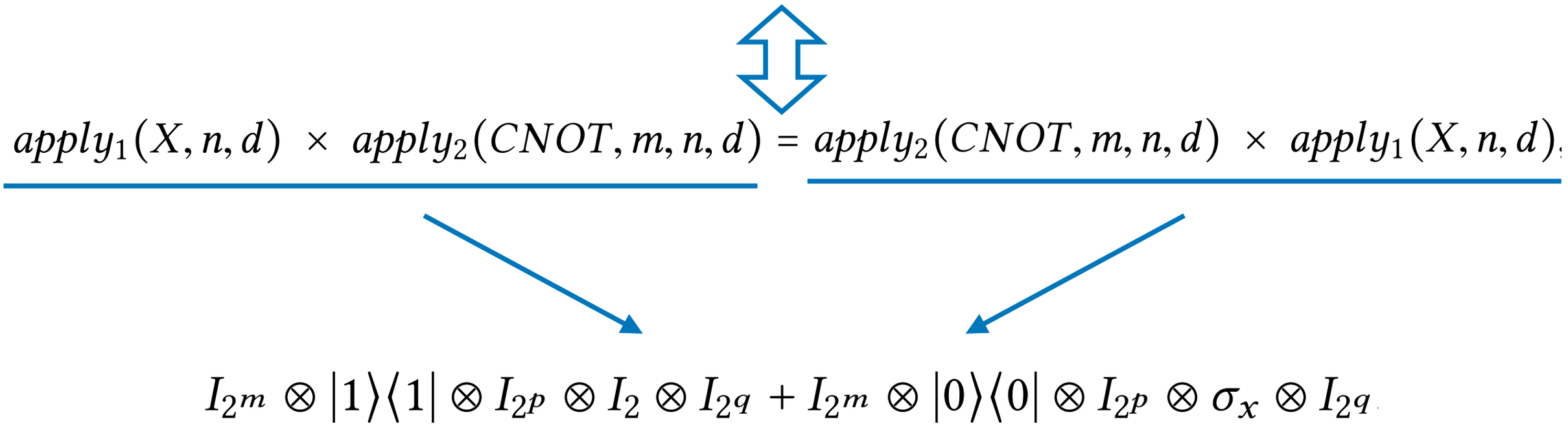
- Proving matrix equivalences in Coq is tedious
- E.g. $X\ n;\ CNOT\ m\ n \equiv CNOT\ m\ n;\ X\ n$

$$apply_1(X, n, d) \times apply_2(CNOT, m, n, d) = apply_2(CNOT, m, n, d) \times apply_1(X, n, d).$$

$$apply_1(X, n, d) = I_{2^n} \otimes \sigma_x \otimes I_{2^q}$$
$$apply_2(CNOT, m, n, d) = I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes \sigma_x \otimes I_{2^q} + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q}$$

Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious
- E.g. $X\ n;\ CNOT\ m\ n \equiv CNOT\ m\ n;\ X\ n$



Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious
- E.g. $X\ n;\ CNOT\ m\ n \equiv CNOT\ m\ n;\ X\ n$

$$apply_1(X, n, d) \times apply_2(CNOT, m, n, d) = apply_2(CNOT, m, n, d) \times apply_1(X, n, d).$$

- Fortunately, this is mostly automated in our development

Experiment

- Evaluated unitary optimizations
- Compared against Qiskit, t|ket⟩, PyZX, Nam et al. and Amy et al.[2]
- Benchmark of 29 programs from Amy et al., ranging from 45 to 61629 gates and 5 to 192 qubits
- Considered reduction in total gate count and T-gate count

[2]Amy et al. *Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning.* 2014.

Results

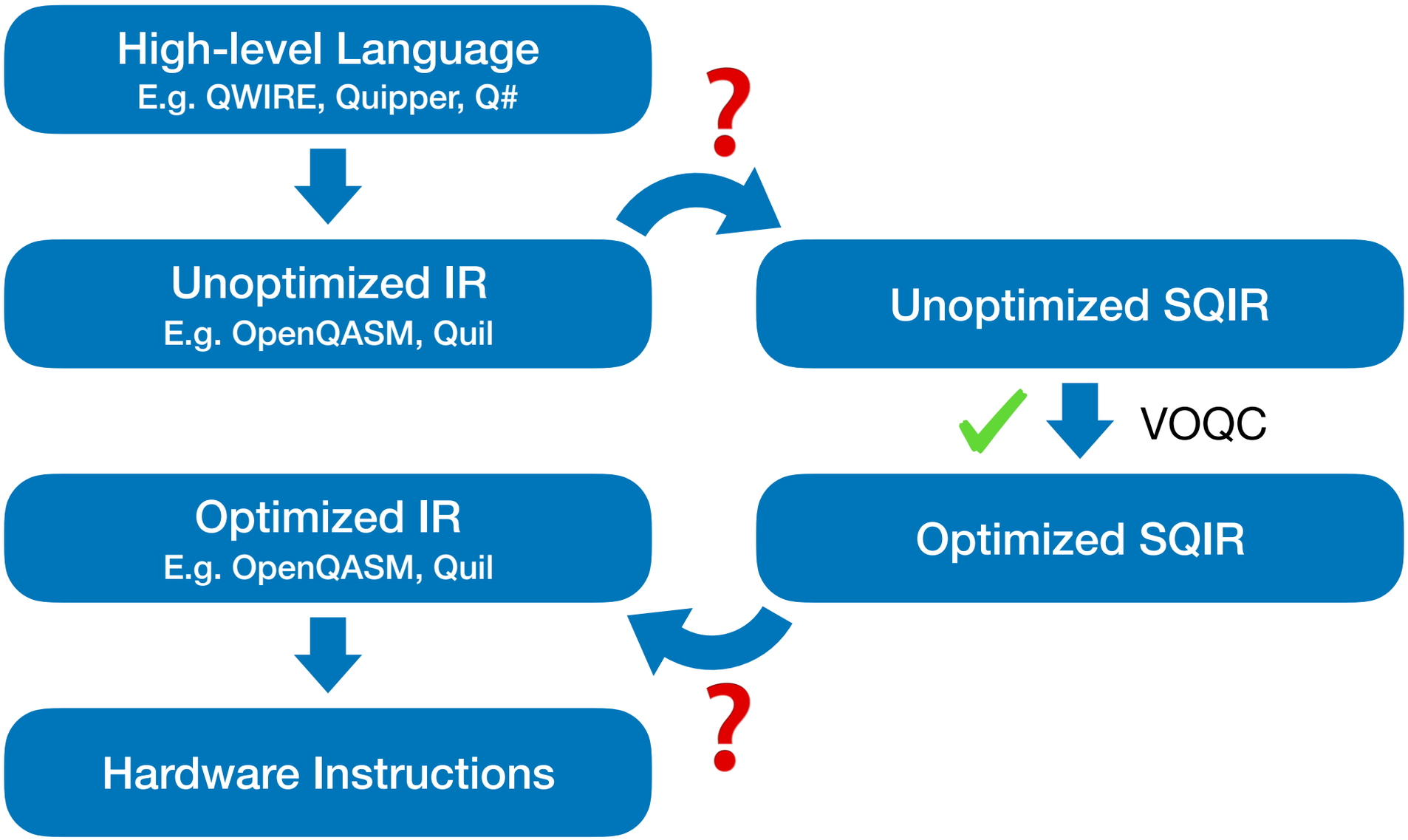
- Average gate count reduction

Nam et al.	**Qiskit**	**t|ket⟩**	**VOQC** ✓
26.5%	10.7%	11.2%	18.4%

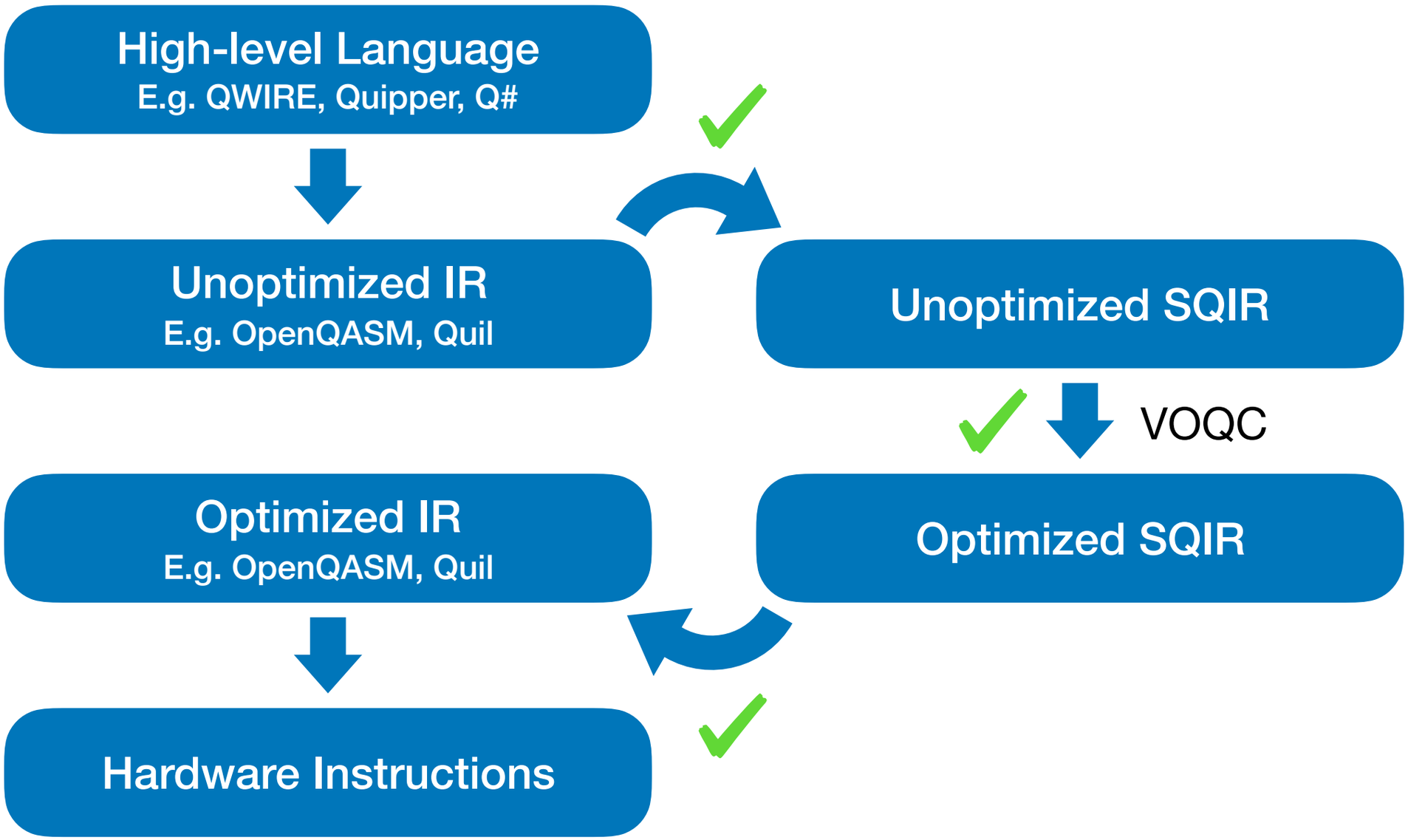
- Average T-count reduction

Amy et al.	**Nam et al.**	**PyZX**	**VOQC** ✓
40.9%	41.0%	42.6%	39.4%

Is the translation between industry IRs and SQIR correct?



Is the translation between industry IRs and SQIR correct?



We verify translation between OpenQASM and SQIR

A feature-complete parser for OpenQASM

Translation between unitary fragments of the two languages

A denotational semantics for unitary OpenQASM

Semantic preservation property of translation

Unitary SQIR and OpenQASM have fairly similar abstract syntax

SQIR

$$U \ ::= U_1;\ U_2 \mid G\ q \mid G\ q_1\ q_2$$

$$G \ ::= H \mid CNOT$$

Qubits are indices into a global register

OpenQASM[a]

Expression	E	::=	$x \mid x[i]$
Unitary Statement	U	::=	$\mathsf{H}(E) \mid \mathsf{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid U_1;\ U_2$
Command	C	::=	$\mathtt{qreg}\ x[i] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \mid U \mid C_1;\ C_2$

OpenQASM is a larger language
that supports declaring qubit registers
and user-defined gates

Unitary SQIR and OpenQASM have fairly similar abstract syntax

SQIR

$$U \ ::= \boxed{U_1;\ U_2} \mid G\ q \mid G\ q_1\ q_2$$

$$G \ ::= H \mid CNOT$$

Qubits are indices into a global register

OpenQASM[a]

Expression	E	::=	$x \mid x[i]$
Unitary Statement	U	::=	$\mathsf{H}(E) \mid \mathsf{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid \boxed{U_1;\ U_2}$
Command	C	::=	$\mathtt{qreg}\ x[i] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \mid U \mid C_1;\ C_2$

OpenQASM is a larger language
that supports declaring qubit registers
and user-defined gates

Unitary SQIR and OpenQASM have fairly similar abstract syntax

SQIR

$$U \ ::= U_1;\ U_2 \mid \boxed{G\ q \mid G\ q_1\ q_2}$$

$$G \ ::= H \mid CNOT$$

Qubits are indices into a global register

OpenQASM[a]

Expression	E	::=	$x \mid x[i]$
Unitary Statement	U	::=	$\boxed{\mathtt{H}(E) \mid \mathtt{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n)} \mid U_1;\ U_2$
Command	C	::=	$\mathtt{qreg}\ x[i] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \mid U \mid C_1;\ C_2$

OpenQASM is a larger language
that supports declaring qubit registers
and user-defined gates

Unitary SQIR and OpenQASM have fairly similar abstract syntax

SQIR

$$U \ ::= U_1;\ U_2 \mid G\ q \mid G\ q_1\ q_2$$

$$G \ ::= H \mid CNOT$$

Qubits are indices into a global register

*Only shown a sample gate set of Hadamard (**H**) and controlled NOT (**CNOT** or **CX**)*

OpenQASM[a]

Expression	E	::=	$x \mid x[i]$
Unitary Statement	U	::=	$\mathtt{H}(E) \mid \mathtt{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid U_1;\ U_2$
Command	C	::=	$\mathtt{qreg}\ x[i] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \mid U \mid C_1;\ C_2$

OpenQASM is a larger language
that supports declaring qubit registers
and user-defined gates

Unitary SQIR and OpenQASM have fairly similar abstract syntax

SQIR

$$U \ ::= U_1;\ U_2 \mid G\ q \mid G\ q_1\ q_2$$

$$G \ ::= H \mid CNOT$$

Qubits are indices into a global register

*Only shown a sample gate set of Hadamard (**H**) and controlled NOT (**CNOT** or **CX**)*

OpenQASM[a]

Expression	E	::=	$x \mid x[i]$
Unitary Statement	U	::=	$\mathtt{H}(E) \mid \mathtt{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid U_1;\ U_2$
Command	C	::=	$\mathtt{qreg}\ x[i] \mid \mathtt{gate}\ x(x_1, \ldots, x_n)\ \{\ U\ \} \mid U \mid C_1;\ C_2$

OpenQASM is a larger language
that supports declaring qubit registers
and user-defined gates

[a]Amy M. *Sized Types for Low-Level Quantum Metaprogramming*. Reversible Computation. RC 2019.

Denotational semantics of SQIR and OpenQASM correspond

SQIR

$$[\![U_1;\ U_2]\!]_d = [\![U_2]\!]_d \times [\![U_1]\!]_d$$

$$[\![G_1\ q]\!]_d = \begin{cases} apply_1(G_1,\ q,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}$$

$$[\![G_2\ q_1\ q_2]\!]_d = \begin{cases} apply_2(G_2,\ q_1,\ q_2,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}$$

A SQIR unitary program denotes a $2^d \times 2^d$ unitary matrix

OpenQASM

Value	V	=	Location, l + Loc. Array, $(l_j, \ldots, l_k)$ + Unitary Gate, $\lambda(x_1, \ldots, x_n).U$
Environment	σ	=	Identifier $\rightarrow$ Value
Quantum State	$\lvert\psi\rangle$	=	2^d-dimension complex vector

$$(\!|-|\!)_E \quad : \quad E \times \sigma \rightarrow V$$

$$(\!|-|\!)_U \quad : \quad U \times \sigma \times |\psi\rangle \rightarrow |\psi'\rangle$$

$$(\!|-|\!)_C \quad : \quad C \times \sigma \times |\psi\rangle \rightarrow \sigma' \times |\psi'\rangle$$

Details elided

Expressions need environment and return bound values

Unitary statements modify quantum state

Commands modify both environment and quantum state

Semantic preservation properties are maintained during translation

$$f : \mathcal{L}(\text{sQIR}) \to \mathcal{L}(\text{OpenQASM})$$
$$g : \mathcal{L}(\text{OpenQASM}) \to \mathcal{L}(\text{sQIR})$$

For all valid programs in SQIR of dimension d, their denotation is equivalent to the denotation of their translation, and vice versa.

$$\forall x \in \mathcal{L}(\text{sQIR}), \quad [\![x]\!]_d = (\!|f(x)|\!)$$
$$\forall y \in \mathcal{L}(\text{OpenQASM}), \quad (\!|y|\!) = [\![g(y)]\!]_d$$

Further, converting from SQIR to OpenQASM and back recovers the original program.

$$\forall x \in \mathcal{L}(\text{sQIR}), \quad g(f(x)) = x$$

Semantic preservation properties are maintained during translation

$$f : \mathcal{L}(\text{sQIR}) \to \mathcal{L}(\text{OpenQASM})$$
$$g : \mathcal{L}(\text{OpenQASM}) \to \mathcal{L}(\text{sQIR})$$

For all valid programs in SQIR of dimension d, their denotation is equivalent to the denotation of their translation, and vice versa.

$$\forall x \in \mathcal{L}(\text{sQIR}), \quad [\![x]\!]_d = (\!|f(x)|\!)$$
$$\forall y \in \mathcal{L}(\text{OpenQASM}), \quad (\!|y|\!) = [\![g(y)]\!]_d$$

Further, converting from SQIR to OpenQASM and back recovers the original program.

$$\forall x \in \mathcal{L}(\text{sQIR}), \quad g(f(x)) = x$$

But the reverse direction does not hold.

OpenQASM parser written in OCaml programming language

Conforms to OpenQASM spec[b]

Uses OCamllex and Menhir parser generator

Available now as an OCaml library on OPAM package repository:

```
opam install openQASM
```

[b]Cross et al. *Open Quantum Assembly Language*. arXiv:1707.03429

Ongoing and Future Work

- We’re always looking for more transformations to verify
  - Optimization from other compilers (incl. error aware)
  - More sophisticated circuit mapping
  - Compilation of classical circuits
- Performance improvements; evaluations on larger sets of benchmarks
- Larger verified toolchain
  - Translation and verification of non-unitary fragments
  - Validate our OpenQASM parser using Menhir’s Coq backend
  - Verify translation from high-level languages such as QWIRE

Conclusions

- We have developed a compiler **VOQC** and an IR **SQIR**, both implemented and verified in Coq
  - Performance is comparable to state-of-the-art compilers
- We have also taken steps to ease interoperability with industry toolchains with translation from and to OpenQASM
- Lots of ongoing work, let us know if you're interested!
- Code:
  - `github.com/inQWIRE/SQIR`
  - `github.com/inQWIRE/openqasm-parser`
- Draft of VOQC paper: `cs.umd.edu/~mwh/papers/voqc-draft.pdf`